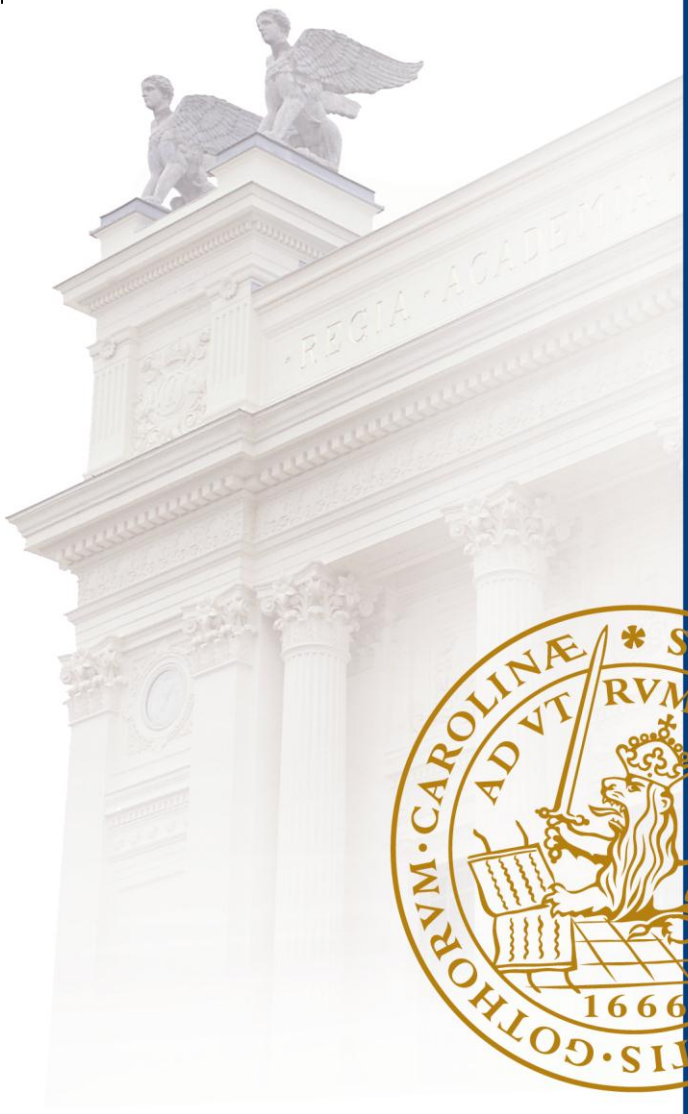
Master's Thesis

A state-based 3-way batch merge algorithm for models serialized in XMI

Aron Lidé

Department of Computer Science
Faculty of Engineering LTH
Lund University, 2011



ISSN 1650-2884
LU-CS-EX: 2011-24

A state-based 3-way batch merge algorithm for models serialized in XMI

Aron Lidé

Supervisor: Lars Bendix

Department of Computer Science
Faculty of Engineering
Lund University

November 2011

Abstract

With the ever-increasing usage of models for development in the industry, the need for merge tools for models to support parallel work on the the same files grows. Models are serialized in XMI, used to represent them as trees with nodes in text form. Merge tools that are intended for text-based development can make the XMI code in the model files syntactically incorrect, making them unable to be opened in their intended editors. There are also conflicts that are not found with text-based merge tools. The model merge tools that exist often require the developer to take care of the conflicts when merging, not creating a merged output until all conflicts are taken care of. We would like to give the opportunity to simply create a file with conflicts and warnings represented to later be fixed.

Due to the structural nature of models and its complexity, the possible combinations of changes to consider when implementing a model merge tool are quite many. There is therefore need for a thorough analysis of these combinations, done in a structured way so as to make sure that all are covered.

In this thesis a 3-way batch model merge algorithm is designed and implemented. The models are analyzed to find changes and inconsistencies before finally being merged. The output needs to be able to represent warnings and conflicts in a way that keeps its XMI valid. The algorithm is based on the algorithm in Martini, A., Merge of models: an XMI approach. The purpose of this thesis is to see if it is correct, if it is possible to implement, what complications that can appear when doing so and how well-covering it is. To this end a test suite was created, which can be used as a basis for other model merge tool test suites.

To Naishi

for her constant love and support

Contents

List of Figures

List of Tables

List of Abbreviations

C	conflict
C1	change 1 (in V1)
C2	change 2 (in V2)
CA	common ancestor (of V1 and V2)
CI	context issue
D1	developer 1 (working on V1)
D2	developer 2 (working on V2)
DD	different destinations (of moves)
DO	different orders
DP	different properties
DV	different values
MD	Move destination
MDD	model-driven development
MOF	Meta-Object Facility
MS	Move source
N1	node 1 (which C1 pertains to)
N2	node 2 (which C2 pertains to)
NA	Node addition
ND	Node deletion
NP	not possible
NR	Node reordering
OMG	Object Management Group
P	possible
P1	property 1 (C1's)
P2	property 2 (C2's)
PA	Property addition
PD	Property deletion
PR	Property reordering
PU	Property update
SC	same change
SO	same order
SP	same property
SV	syntax violation / same value
UC	unit of comparison
UML	Unified Modeling Language
V	version (arbitrary)
V1	version 1
V2	version 2
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1

Introduction

When working in a group of developers, you don't want to lock the files from being edited by more than one developer at a time. To this end the developers will have to copy the files in the system to their own workspace where they can work on them by themselves. This is not a problem, but when the developers need to commit the changes they've made to the system into the shared version of the system, in the repository, the files need to be merged with the new version of the system in the repository as other developers have commited their changes. There are different ways to merge, but most of them look at the two versions and their common ancestor to see what different changes there have been in the different versions and then make those changes as long as there isn't a conflict when the developers have changed the same lines, in which case both changes are represented in the text and the developer has to manually fix the conflict.

Since software development traditionally is done in text form, almost all merge tools are developed with text file merging in mind. However, with the ever-increasing usage of UML and other models for development in the industry, the need for merge tools especially designed for merging models has increased. This is because merge tools that are intended for text-based development can make the model files syntactically incorrect, making them unable to be opened in their intended editors. Models are serialized into text form according to XMI, an XML dialect, which describes this syntax. The structural nature of models makes it easier to find connections between different parts of the files than in normal code, but at the same time it makes it harder to make the merged files syntactically and semantically correct, especially when giving the developer options from both of the versions in a conflict. Often, model merge tools require the developer to take care of the conflicts when merging, in an order decided by the tool, before there is created a merged output file. But when merging, and especially when merging with models, there is often need for the developer to look at many changes that are connected before deciding what to do. Therefore we would like to give a batch merge solution, meaning that the algorithm doesn't get any input from the user. The merged output file will therefore have conflicts and warnings represented in it. The developer can then take care of the conflicts with the flexibility to do it in whichever order he wants, and whenever he wants.

In this thesis we will analyze, design, implement and test a 3-way batch merge algorithm for models, which means that the algorithm takes two versions

of the same model file together with their common ancestor and without any input from the user creates a merged output. The two versions will be analyzed to find changes, which are compared to find conflicts before finally being merged as a valid XMI file. The algorithm is based on the algorithm in Martini [10].

We will analyze the structure of the models to discern all the different kinds of changes that can be done in a model, what distinguishes them, how they affect each other and how to best take care of them in case they end up in conflict with each other. With this as basis we can in a structured way study the possible combinations of changes and make sure that the algorithm takes care of them in a proper way. This is done to make sure that the merge tool covers as many kinds of merges as possible. These combinations can also be used as the basis for a test suite for the tool, and for other model merge tool test suites as well.

We will design a more detailed algorithm than Martini's and will then try to implement the algorithm to see if it's possible to implement and the problems that appear when doing so. In the end we will also test how well-covering the algorithm is with the test suite.

Chapter 2

Background

In this chapter we will provide the background and context needed for the later parts of the thesis. First, since we are working with models, we need to introduce models and their structure as well as why and how they are used in development. We will also look at how they are serialized in XMI and how XMI is structured. Secondly, the problem of version control and the need for merge tools when working in parallel will be explained. The different kinds of merge tools will be presented and why traditional merge tools don't work for models. This will give insight to the general problem dealt with in the thesis. Finally, we will give an overview of the algorithm presented in Martini [10] to lead into the later parts of the thesis where analysis, design and implementation of this algorithm will be done.

2.1 XMI and models

Models have been used in software development for a long time, mostly for designing products, but lately also for auto-generating code directly from the models. There are a lot of different types of models, but they all use the same standard for exchanging metadata information. This is the XML Metadata Interchange (XMI) specification from the Object Management Group (OMG). XMI is a standardized way of using XML to serialize models. Although there are more types of models we will for the sake of simplicity use the most common model type for discussion, the Unified Modeling Language (UML), which also is an OMG standard.

Every model that follows the OMG standard conforms to a metamodel which defines the structure and syntax that is used in the model. Since a metamodel also is a model, it in turn has to conform to a metamodel, a meta-metamodel. Meta-Object Facility (MOF) defines a model which conforms to itself, being the metamodel for itself. Because of this MOF is on the highest level of abstraction, an M3-model. Every model type (like UML) has a metamodel that conforms to MOF, and every model of that type conforms to its metamodel, and their run-time instances conform to the model, see figure 2.1. This forms a hierarchy where the higher layers are more abstract than the lower ones. Every serilized model instance that conforms to a meta-model that conforms to MOF indirectly follows to the XMI standard. A model that is serialized in XMI correctly is *XMI*

valid.

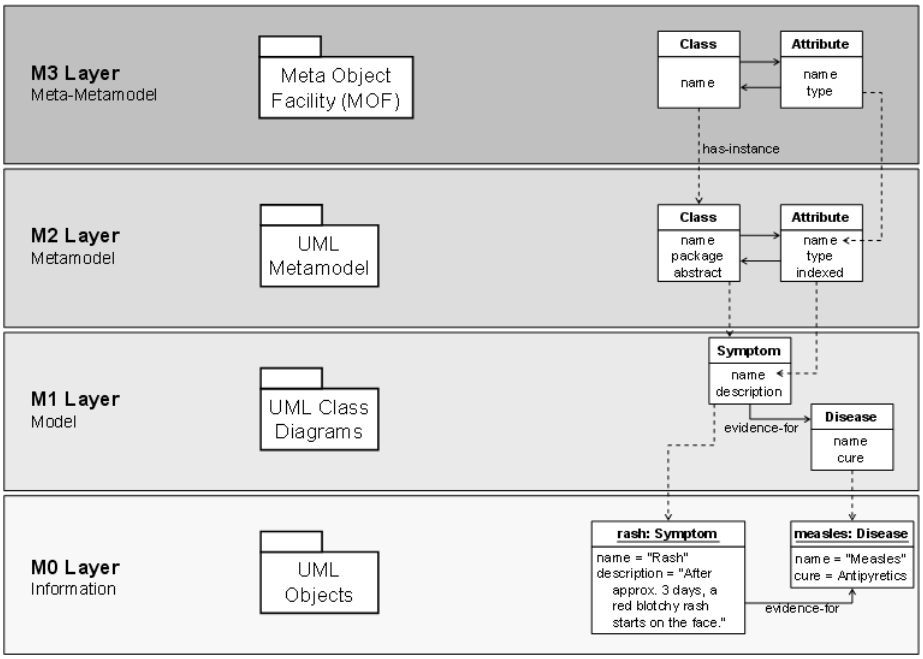


Figure 2.1: *The four-layer metamodel hierarchy for UML.*

Each of the models (layers M1-M3) can be serialized in XMI. The models in layer M2 and M3 are abstract models (metamodels) which represent the semantic information, whereas the model on layer M1 is a concrete model. The concrete model needs to be serialized in XMI when saved, although the extension on the file isn't always `.xmi`, as is the case with UML files.

Even though XMI is a standard for serializing models, there exist different specifications of XMI, and unfortunately they are not completely compatible with each other. In this thesis we have chosen to use the XMI 2.1 specification [12] due to the fact that the tool used for creating the models used during implementation uses that specification.

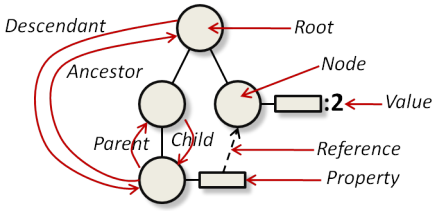


Figure 2.2: *Different parts of the XMI model.*

An XMI file, as every XML file, is structured as a tree, see figure 2.2. The tree consists of a number of *nodes* that are connected. Every node has zero or more *child nodes*, which are located below it in the tree. The child nodes of a node are each others' *siblings* and they have a specific order. A node that

has a child node is called the child's *parent node*. Every node has exactly one parent, except the *root node*, which has none. The root node is located at the top of the tree, and has paths down to every node in the tree. The path from the root to a node is called the node's *root path* and the length of it the *depth* of the node. The root node thus has the depth 0. All the nodes that are located below a node are called its *descendants* and all the nodes in a node's root path, including the root, are called its *ancestors*. Any node in the tree is a root to a *subtree* which includes itself and all its descendants.

Every node can have one or more *properties*. Every property has a *name* which is unique among the properties in the node, and a *value* which is represented by a simple string. Just like a node's child nodes are in a specific order, so are the properties. The property `xmi:id` or `xmi:uuid` contains the ID of the node, which in the former case is unique in the tree and in the latter case globally unique. The value of a property in a node N1 can contain the ID of a node N2, in which case the property points to N2, creating a *reference* between N1 and N2. The ID is also, as we will see in 3.1, important for being able to identify and access the nodes without depending on the path to the node. IDs for the nodes are strongly recommended to have, but not compulsory.

In an XMI file the root node is an *XMI element*, which denotes the start of XMI information and identifies the XMI version that is used.[1] The XMI element can have many child nodes, but we are mostly interested in the *model node*, which contains the actual model. Many times in XMI files there is only this single child node of the XMI element, in which case the XMI element itself doesn't need to be present (as it's not in figure 2.3). In this case, the XMI version that is used needs to be specified in the model node [12]. From now on, when we talk about the "model", we refer to the the tree in the XMI file with the model node as the root.

Figure 2.3 shows how a simple UML model can be serialized in XMI, and the model in between is the graph representation of the XMI serialization. The coloured lines connect the parts that represent the same things. As we can see, the model node has properties that declare that XMI 2.1 and UML2 3.0.0 is used for this model.

UML models (and other models) can be serialized using XMI in different ways. The serialization pattern that is going to be used in this thesis doesn't allow nested entities, that is nodes of the same type under each other. Different kinds of nodes in the model can only be located at a certain depth from the root node, the model node. Child nodes of the root can represent classifiers, associations, etc. These are connected in the model using references, which are always made to nodes at depth 1. Child nodes of classifiers can be attributes and operations and child nodes of association can be association ends, which have references to the classifiers the association connects. Using this serialization pattern the model never gets very deep, but gets broader instead, which makes it easier to traverse.

It is also possible to serialize models in XMI in a way that allows nested entities. This means that a node type can exist on many different depths, which we will see will cause some trouble later in 3.2.3. Using this pattern one

[1]The XMI element can be present in any XML file of which not the whole conforms to the XMI standard. This is to show that the subtree with the XMI element as the root node does conform to it.

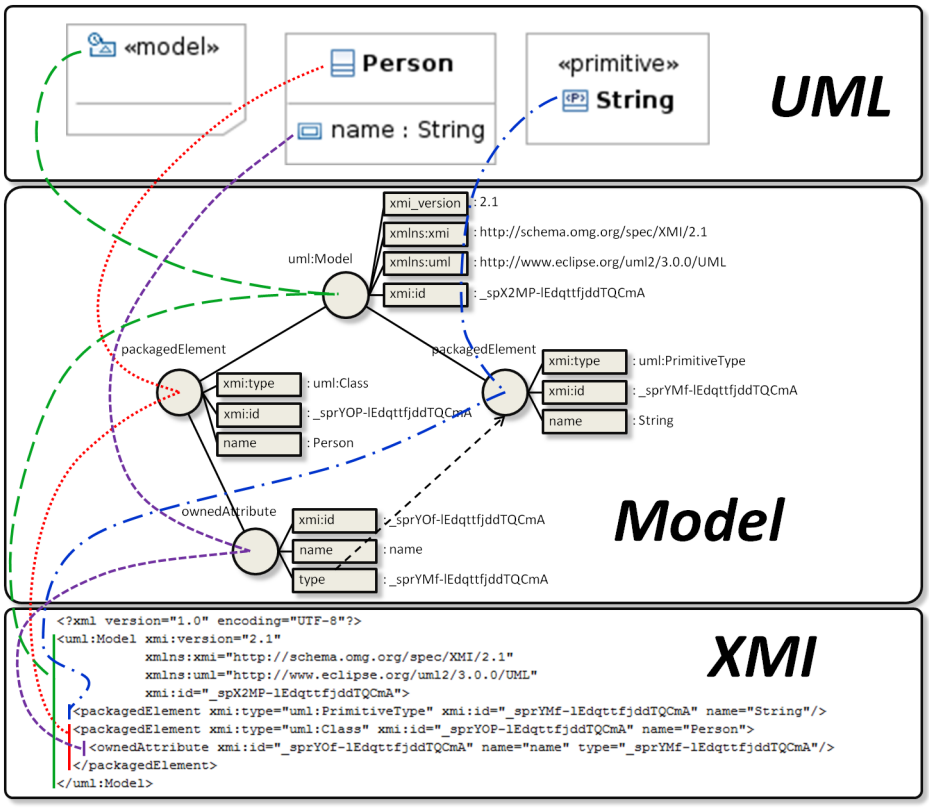


Figure 2.3: *A small UML model represented as a model and serialized in XMI.*

classifier can have another classifier as its child node because they are connected in a certain way, like one of them being the subclass of the other. Connections between different nodes are therefore not as often represented by references as in the previous pattern (the one that we will be using).

2.2 Version control

Usage of models in the industry is very common and as it's starting to be used more often as a part of the implementation when used for auto-generation of code, instead of just for designing the architecture of the software, changes to the models are of much greater importance and happen more frequently. This is because there are not only designers, but also developers working on the files, and contrary from how it was before, the models have a direct impact on the code of the system. Therefore parallel work on models has increased, which means that there is a greater need for tools to handle the problems that come with that way of working.

When a group works on the same files in parallel using version control there exists a shared repository in which the files of the system are stored. The developers copy these to their own workspace (*check-out*) and make changes which they *commit* to the repository after checking that their code is not faulty.

However, if there has been a commit from a developer **D1** since developer **D2** checked-out, the code in the repository contains D1's changes which need to be added to D2's version of the system before it can be committed to the shared repository. Developers thus have to *update* their workspace with changes from the repository before committing, which, if D1 and D2 have been working on the same files, means that these files need to be *merged*. See figure 2.4.

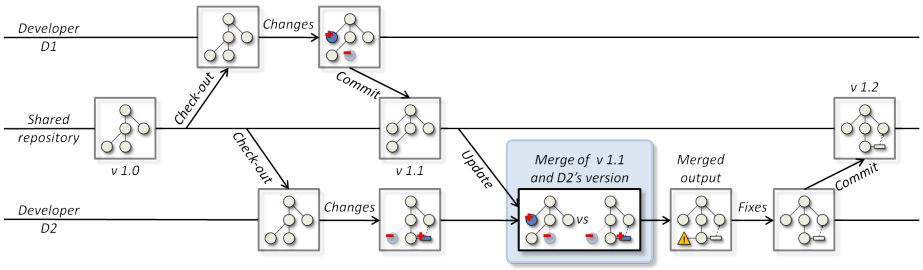


Figure 2.4: *Parallel work between developers D1 and D2 leads to a merge.*

The usual way of merging is called a 3-way merge and takes the *two versions* (V1 and V2) of the files and their *common ancestor* (CA) to find what changes have been made in the two versions and apply these changes to a copy of the CA. If there are any changes in the two versions that contradict each other, then there is a conflict. In text-based development the merge tools most often check if changes have been made on the same lines, in which case the two options are represented in the file one after the other. The developer can then choose one of them or make a new solution, either manually or by help from the merge tool.

There are two problems which make it impossible to use text-based merge tools for models. First, because of the structure of models, two changes in two different parts of the file can very well be in conflict due to references and moves in the model.[2] Secondly, since models are serialized in XMI there is no possibility to merge like this for models, lest the files should become invalid and impossible to open in model editors. The merge tool needs to apply changes and represent the options for the developer when there are conflicts with the XMI syntax in mind to not break it. This will pose problems as conflicting changes can't be naturally represented in accordance with the XMI syntax.

There are a few things that distinguish different merge tools from each other. The one described above is a **state-based** 3-way merge, meaning it compares three files, the two versions and their CA, which are in a certain state. There is an alternative to the state-based approach — the **operation-based** approach [8] — which relies on the editor to record the operations that have been made on the files so that the sequences of operations from the different versions can be merged into one sequence of operations which then later is used to create the merged file. This approach is further discussed in 6.2.

There are also differences in how merge tools handle conflicts, where a lot of tools (especially model merge tools) require **interaction** from the developers, forcing them to take care of conflicts during the merge process, before a merged output can be created. The alternatives in conflicts are often presented for the

[2]There are similar problems in text-based merging too, where two changes on different lines are incompatible.

developer to choose one or the other, and there is a problem of in many cases not being able to create an own solution. Moreover, the developer is often forced to take care of the conflicts in a certain order, decided by the tool. Since there often are a lot of connections between changes in models, it poses a problem for the developer to take care of conflicts when he's not having an overview of all the changes done to the files. This can lead to files which may be syntactically correct, but not logically and semantically correct. This is a major drawback, especially since the developer might think that the merge is okay, because the files can be opened in the editor.

The traditional way of merging, which was described for text-based merge tools above, creates merged output files in which all the changes are made and the conflicts are represented, so the developer can take care of them after the merge process is over. This is called a **batch merge**, and is not yet that well spread for model merge tools. A reason for this is the apparent problem of representing conflicts in a syntactically correct way in models. The advantage that you get from this way of merging is that the developer can get an overview of all the changes that are done in the two versions and where they are in conflict to better understand how to solve them. It also gives the developer the flexibility of choosing when to take care of the conflicts, while other tools require the developer to be active throughout the whole merge process, which can be long and complicated, before there is a merged output.

2.3 Algorithm overview

We will be looking into the work done by Martini in [10], where a state-based 3-way batch merge algorithm for models serialized in XMI was presented. The goal was to create a merge algorithm which is independent from both the model type (such as UML, Ecore, etc.) as well as the editors used (which the operation-based approach is dependent on). The choice was made to merge on the XMI level to stay independent from the model type, and therefore no semantics from the model could be used, but only the semantics extracted from the XMI structure. Staying independent from the editors meant only using information taken from the files themselves. This also meant that the delivered output should not be specifically designed for a certain model type or editor.

Martini's merge algorithm consists of five parts, which will be briefly described here. They are *change detection*, *conflict detection*, *change interpretation*, *merge rules definitions* and *change application*. Figure 2.5 shows the flow of the algorithm.

Change detection is the act of extracting changes done in the two different versions, with respect to their CA. This procedure was created by first analyzing how the XMI model can be changed, defining different kinds of changes, their nature and how much of the model they cover as well as what is needed to represent them. Secondly, it was shown how to compare the model in the new version with the one in the CA by traversing them in a certain order to catch all the changes.

Conflict detection is done by comparing the changes in two versions to see if they change the same item in the model, and if so, if it is done differently

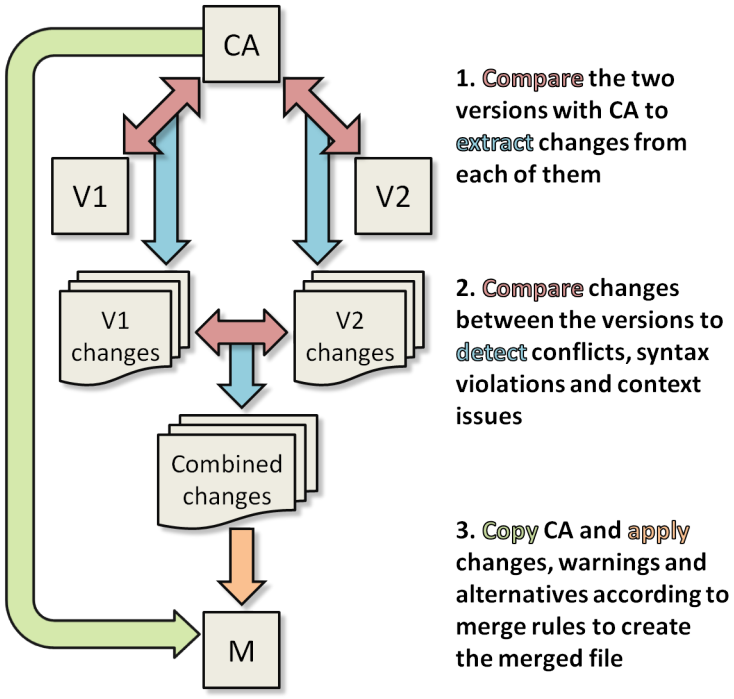


Figure 2.5: *The merge process.*

in the two versions. Because of the nature of the different kinds of changes the comparison is done in an order dependent on the types of changes.

Change interpretation is an in-depth conflict detection analysis of changes that are not in a direct conflict, but indirectly could lead to either violations of the XMI syntax or probable context issues. This is done by once again comparing changes with each other to see if they affect each other and break the syntax, or if they are sufficiently much connected to each other to be causing some kind of context related problem.

Merge rules definitions describe how to best take care of conflicts, syntax violations and context related problems. The solutions presented are to discard one or more of the changes in conflict, and/or to give the developer a warning about the situation, always with the requirement to not lose any information in mind.

Change application is the part that describes how to apply the changes to the copy of the CA to create the merged output. An analysis of the order in which the changes need to be applied to the model was done, making sure that there are no problems when applying later changes. Suggestions on how to represent annotations in the model, such as warnings and alternatives in conflicts, were also presented.

In his final algorithm, Martini combined the conflict detection, change interpretation and merge rules definitions parts to take care of all the conflicts

and syntax violations at the same time. He presented a concise algorithm which gave a good overview of what should be done, and generally in which order to do so.

This thesis will supplement Martini's work, theoretically as well as an empirically. First, we will further analyze the changes that can be made to the XMI model to cover all possible combinations of changes that can be done to both one and two versions of a model, which, as we will see, was not done in Martini's work. We will also introduce a new type of change, which was considered of lesser importance to Martini, but which we will show makes a difference, and which we will include in our analysis and algorithm. Our structured analysis will also give the outline for a test suite that will try to cover all possible combinations of changes.

Secondly, since Martini didn't have time to implement and test his algorithm, we will do so in this thesis and present both the problems that were encountered during implementation and the results of the testing. The algorithm presented in this thesis will be more detailed and easy to follow and will describe exactly which version of the model information should be taken from and in what order.

Chapter 3

Analysis

In this chapter we will analyze the models and create a requirement specification which will be used in the next chapter where we will design the algorithm. This is the basis of our whole thesis, and we will introduce new terms and vocabulary that are needed to understand the rest of the thesis.

First, some basic requirements which will affect our later analysis will be presented (3.1). After that we will analyze and describe changes on the model, their attributes and types (3.2). We will then cover the combinations of changes that can be done in one version (3.3.2) as well as in two versions (3.3.3), which will present the different types of conflicts and syntax violations that can appear, and we will analyze how to best deal with them (3.4). Then we will analyze the problems in the actual merging process where changes are applied, such as conflict representation (3.5). Lastly, we will analyze how to arrange a test suite for testing of the merge tool (3.6).

3.1 Initial requirements

Here we will present a few requirements that we need to satisfy in order to create the kind of merged output that we wish to have. We will strive to satisfy these requirements as we analyze the model throughout this chapter, and requirements based on this chapter will be presented in 4.1.

The most basic requirement we have is that both the input and the output files should be **XMI valid**. If the input files are not valid, they can't be guaranteed to be processed correctly; if the output files are not valid, the user won't be able to open them in their editor. Thus, changes that in combination create syntax violations should be handled in a way that these are removed.

To be able to compare versions of the same model with each other, the elements in the model need to make identified to be sure that the correct parts are compared. We need to be able to match the elements of one version to their corresponding elements in the other version (given that they exist). This can be done either by using computationally expensive and error prone algorithms which match the models by traversing them and comparing the different parts, or by giving every element an ID in which case we match by the IDs. In the rest of the thesis we will assume that **every node in the model tree has an ID** to represent them, both for the sake of comparison and to be able to have

references, which depend on IDs.

When a developer looks at the merged output he needs to be able to get as much information as possible about the changes that have been made and the conflicts that are present. Therefore, we require **no loss of data** from the merge. This requirement can be split into a few parts. First, the tool should be able to find all the changes made in both versions to not lose any change. Secondly, in cases of conflicts or syntax violations, we need to keep all the data that represent the different options presented in the two versions for the developer to work with. Thirdly, we should inform the developer about the changes that have been made which were not in conflict with any other changes, simply for him to be able to consider the impact they make on the whole merge.

We also require **symmetry** of the merged output. The output should always look the same if the input files are the same, even if the two versions are in different order. The changes in the versions should therefore always be presented with the same amount of priority, since we can't know if one version is more "correct" than the other.

3.2 Changes

To be able to thoroughly and correctly examine how to compare the models and find conflicts and violations we need to understand what kinds of changes that can be made on the models, what differentiates them from each other and what information we need to describe them.

First we will introduce some characteristics and vocabulary of changes to be used in later discussion. Then we will specify the attributes of changes that we need to represent them correctly and differentiate between them. Using these we will then discuss the different types of changes and their interaction.

3.2.1 Change characteristics

There are two kinds of changes that can be made on a model. Models consist of nodes and properties, so we can have changes that pertain to either one of them; **structural changes** are changes made to the nodes and the structure of the model and **data changes** are changes made to the properties, which contain all the data in the models [1]. Properties have no structural nature since they contain only a simple string. They can only affect one part of the model and are therefore called **simple changes**. Nodes, on the other hand, have child nodes, who themselves can have child nodes, and a change on a node can affect all the nodes under it. If, for example, a node is deleted, all the nodes under it are also deleted. Changes like these are called **composite changes** because they compound many changes of the same type on many nodes.

As we can see, different kinds of changes affect different areas of the model. There can be two changes (one in each version) made on the same node without them being in conflict, because they are changes of two different properties. At the same time two changes on two different nodes can be in conflict because one of them covers many nodes, including the node the other change is made to. We need to localize the changes as deeply as possible to see exactly which parts that are affected to make the changes as independent as possible. Each change will cover a certain "area" of the model, which will be their **units of**

comparison (UC). UCs are presented in Oliviera, et al. [11], and are defined as "an atomic element used for conflict computation. Conflicts occur when two or more developers concurrently work on any part of the same unit of comparison" (p. 3 [11]). In normal text-based merge tools the UCs are the lines, and in the case of XMI we have the nodes and properties, the nodes having their subtrees as part of their UC.

If two changes in different versions affect the same UC we have a conflict, and since we don't have any conflicts in one version one could draw the conclusion that UCs of changes made in one version can not overlap, meaning that any item in the model is never directly affected by more than one change. This is true in most cases, but we will see some special cases in 3.3.2. We need more information about the change types before we can discuss those.

We want to extract from the models all the information we need about the changes to be able to compare them later without needing to access the model again. This is because it's easier to look for conflicts if we have all the information already at hand and because we want to separate change detection and conflict detection parts of the algorithm as much as possible.

3.2.2 Change attributes

Here we go through the attributes we need to represent a change, why we need them and what they mean. We will see that there exist some extra attributes, but these are specific for each type of change and will be presented under their respective section in 3.2.3.

The version that a change is made in is the first, and most obvious attribute. A change can only be in conflict with a change in the other version, since we use XMI valid input.

The position of the change is used to (partly) describe the UC of the change. Since every change is made to a specific node, the ID of that node is used to represent the position of the change.

The type of a change simply states the type of change that has been made, for example a deletion of a node or an update of a property value. We discuss the types further in 3.2.3.

The name of the property is a further specification of the UC of the change, to point out which property the change is on. This attribute is only needed for the data changes.

The value of the property is also only needed for the data changes and simply provides the new string value of the changed property. It's needed for comparison and application, but what the string actually contains is of little concern, unless the property is a reference, in which case there is a connection to another node which we need to take into consideration.

These attributes are of the nature that when we look at them from the first to the last one we get an increasing accuracy of where the change has taken place and what it entails. First we get to know which version the change is in, then the node in that version, then the type, which decides if we need to go deeper, then the property and lastly the value of the property. If we compare

two random changes, this is the order we would go about to see if they are in conflict, because the way we analyze the later attributes depends on what the earlier ones are. For example, knowing that two changes are changing properties with the same name would be of little help for finding out if the changes are in conflict if the changes are in different nodes, or in the same version.

An additional aspect of the position attribute is the position in order of a node among its parent's child nodes. This is needed to be able, e.g. in the case of an addition, to insert the node at the correct position in order. Similarly, we need to be able to insert properties at the correct position in order among the properties of a node. This can't be easily represented by e.g. using the index number of the node or property, for various reasons, and this is discussed further under *Order management* in the end of 4.2.4.

There are also a couple of other attributes that we want for each change. These are not describing the change as the earlier ones, but are of a more practical nature.

ID We decide to have an ID for each change. This will be used in 3.5 where we need to refer to specific changes.

Application As we will see in 3.4, some changes will not be applied in the end. This attribute says if the change is to be applied, and its default value is true.

We will discuss the attributes more in 3.3.1.

3.2.3 Change types

We will now go into more detail about the different kinds of changes. We will discuss the structural changes first and then the data changes. Nodes can be changed by *adding*, *deleting* or *moving* them. Properties can be changed

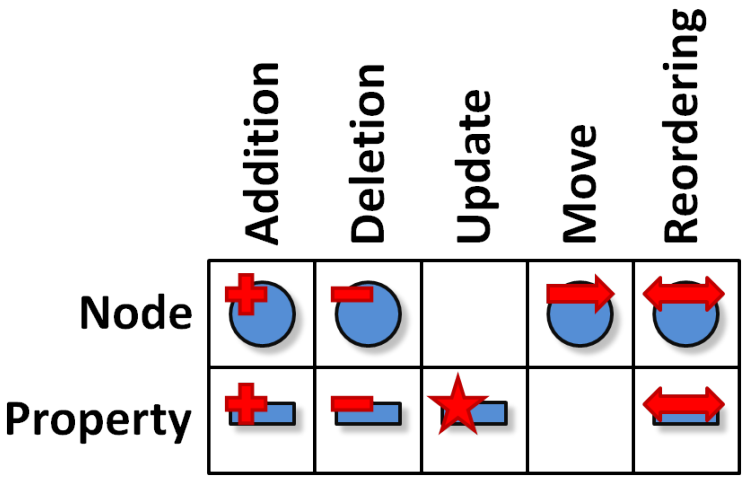


Figure 3.1: *The different types of changes.*

by *adding*, *deleting* or *updating* them. Both nodes and properties can also be *reordered*. See figure 3.1.

Looking at the structure of the XMI model we can deduce that these are all possible types of changes. Except adding or removing any item (node or property), we can only change an item's value or position. A property can only have its data value updated since its name identifies it, and renaming a property would be equivalent to deleting the property and adding a new property with the new name. A node doesn't have any value to be changed, but it has structural connections to other nodes above and under itself and to its properties. Above we can change the parent, which is the act of moving the node. Below, we have the child nodes and the only way to change the structure of the child nodes without changing their connection to their parent is changing their order. The same reasoning applies to the changing the order of the properties.

Node addition

Will henceforth be referred to as **NA**. See figure 3.2.

An NA is a composite change which describes the addition of a node and its subtree. The UC of the NA contains all the parts of the model that have been added due to the NA. This type of change is quite safe in terms of conflicts, since the subtree didn't exist in the CA.

Which kinds the elements that are added are and what they contain doesn't matter much, except in two cases. The only types of elements that can be added in an NA that has any direct connection to other parts of the model has to do with references. First, and most important, we can add a reference property, which will be analyzed closer in 3.3.1. These references should be saved for later comparison with other changes. Secondly, we can add a node that is referenced to from another node in a different subtree. Given the fact that this node is added in the version V, we know that it doesn't exist in CA and therefore whichever reference that is made to it must have been created in V. Thus we will find this new reference later, and it doesn't need to be saved at this moment. We can't actually even know of its existence by simply looking at the elements in NA, so we don't have a choice in the matter anyway.

To keep track of the UC of the change correctly we need more than the ID of the added node, however. If a subtree has been added, there is the possibility that other nodes have been moved under it. The UC of the NA should not include these nodes because they were not added by the NA, and therefore we need to keep track of the roots of the subtrees that have been moved under an

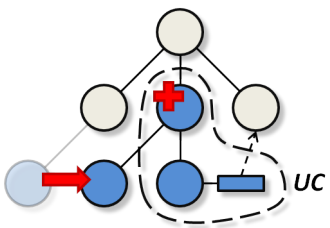


Figure 3.2: *A node addition. The UC of the NA does not include the node moved under it or the referenced node.*

added node in the NA. This can be done with an additional attribute, a list of the IDs of these nodes. We will need this information to make sure that we apply the change correctly.

Node deletion

Will henceforth be referred to as **ND**. See figure 3.3.

An ND is much like an NA, it is a composite change that describes the deletion of a node and its subtree. Contrary to NAs, with NDs we lose a lot of information that is in the CA, which makes it a change type that can end up in conflict with most other kinds of changes.

The only change that can be made under an ND in the same version is a move. We need to keep track of the nodes that have been moved from the deleted subtree to some other part of the model. Just like with NA, we can do this by adding a list of the IDs of these nodes as an additional attribute. We will need this information when we compare changes later in 3.3.3.

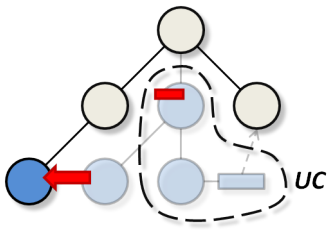


Figure 3.3: *A node deletion. The UC of the ND does not include the node that is moved from under it.*

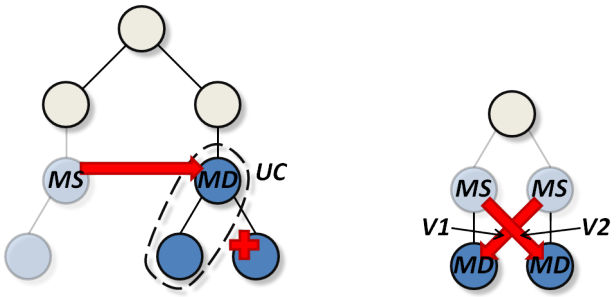
Move

Moving a node is the act of changing its parent. It is a composite change, but not in the same manner as adding or deleting a node. Moving a node only changes the path from the model root to the root of a subtree,[1] not the structure under it or its ID, which means that it doesn't directly affect the subtree, not even the root node of the subtree. It is still a composite change, because it does move the whole subtree, not just the root.

Since moving is the change of position of a node, a move has two positions. As we will be discussing a lot about positions of changes relative to each other, we introduce two different terms to specify which position we mean. The *move source* (**MS**) is the position of the node in CA, and the *move destination* (**MD**) is the position of the node in the new version. See figure 3.4a.

The UC of a move is a bit difficult to define. While the only thing that is actually changed in the model is the edge from one node to its parent, it affects the whole subtree, and we would therefore like to include the whole moved subtree in the UC. As with NAs and NDs, there are cases where further changes to the moved subtree leads to that not the whole subtree is moved to the new parent, like when there is a deletion or a move of a node from a moved

[1] And therefore the path from the model root to every node in that subtree.



(a) *A move of a subtree. The UC of the move does not include nodes that have been added or moved under it.*

(b) *Nested moves. This is not possible with the chosen XMI serialization pattern.*

Figure 3.4: *Moves.*

subtree. Likewise, nodes that are added or moved to the subtree are not either part of the UC of the move. Also, for moves we need to save the IDs of the root nodes of the subtrees that are not part of the UC. Since the moved subtree will be present in both the changed version and CA, making any type of change under a moved node is possible, not just moves as under NAs and NDs.

Moves can not pertain to properties, as moving one property from a node to another one would mean deleting it from the source node and adding a new one to the destination node. This is because a property does not have an ID of itself, but relies on the ID of the node and its name for identification. Having properties with the same names in different nodes is not only possible, but also extremely common. The normal node move change could however not be represented as a deletion and an addition as it would mean that changes made under the moved node would not be noticed and only incorporated as part of the newly added subtree. Therefore we need this type of change.

Because of the way we have chosen to serialize the model, a certain node type can only exist at a certain depth from the model root.[2] This leads to that nodes can not be moved vertically, the only way to move a node is laterally, to a parent at the same level as the previous parent. This makes it impossible to make nested moves, where you need to be able to move to other levels for it to be possible. An example is if you in one version move a node X under another node Y, and in the other version move Y under X, see figure 3.4b. We will see more advantages of this serialization pattern later.

Property addition

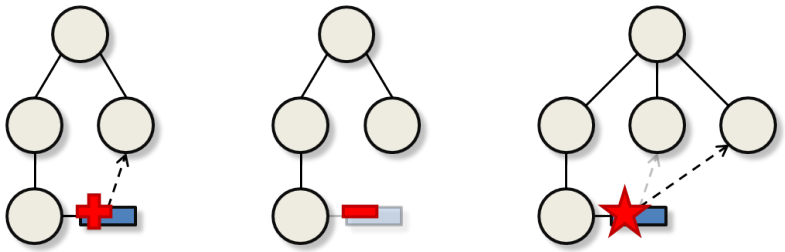
Will henceforth be referred to as **PA**. See figure 3.5a.

A PA is a data change which describes the addition of a property in a node. Its UC is just the property in question. As the case is with NAs, PAs can lead to a new reference, which we need to save for comparison with other changes.

Property deletion

Will henceforth be referred to as **PD**. See figure 3.5b.

[2]E.g. MOF classifiers and associations can only be child nodes of the root.



(a) *A property addition leading to a new reference.*

(b) *A property deletion.*

(c) *An update leading to a changed reference. Only the value is part of the UC.*

Figure 3.5: *Data changes.*

Just like PA, a PD is a data change. It describes the deletion of a property in a node. This is probably the easiest change to take care of as it's a simple change with a small UC and it doesn't lead to any added references.

Property update

Will henceforth be referred to as **PU**. See figure 3.5c.

A PU is simply an update of the value of a property. It can, like NAs and PAs, lead to a new reference that we need to save. A PU only changes the value, unlike PAs and PDs, which change the whole property. This means that the only thing that is part of a PU's UC is the value of the property.

Node reordering and Property reordering

Will henceforth be referred to as **NR** and **PR**. See figure 3.6.

As we've made a distinction between the addition (or deletion) of a node and that of a property, we will make the same distinction here between reordering child nodes and properties of a node, but present them together because of their similar nature. There is, however, an important distinction between PRs and the other changes pertaining to properties, and that is that PRs are not data changes since they don't affect the data of the properties.

We have covered the changes that are described in Martini's thesis [10], but we will look into how the change of order affects the model. Martini wrote that "we do not need to consider child-node order" (p. 55) with the argument that our model is not an ordered tree, like that of Lindholm [9]. We would, however, not like to reject any changes in order that have been done, because we think that if such changes have been made, there are a reasons for it, and we should consider them. Changing the order of properties is not always possible with some editors, but we do not want to be dependent on the editors.

We know that reordering child nodes or properties does not have any impact on the final system product, just like the order of the attributes and methods in a class doesn't matter. But there will be a difference in the model which we are working with. On the lowest levels, looking at the XMI and the model view, such changes are seen directly. On higher levels, like in the class diagram of a UML model, the changes are not always seen, but they can be. For example, the

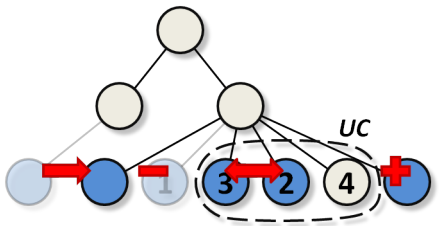
order in which the attributes in a class are presented is the order the attribute nodes have under the class node in the model.

The fact that the order of child nodes and properties does not have any impact on the final system product is the main reason that these changes seem to be considered less important. This leads us to wonder about how we should prioritize this kind of change. If two child nodes are reordered in one version and one of them is deleted in the other version, would that be considered a conflict? One could prioritize NRs and PRs as less important, which would mean that they could only be in conflict with a change of their own kind. However, we can't know that the change of order is something that's not as important as the other changes, and especially not for all possible cases. We will therefore prioritize reorderings the same as other changes.

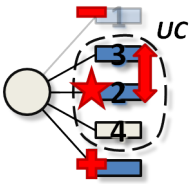
What do we consider as changing the order, then? If we add, delete or move child nodes to or from a node it has an impact on the order of the nodes, but it's not because of an order change. Therefore we don't want to consider the changes in order that those changes impart when looking for NRs and PRs. If a node has been added or moved under a node N in a version V, it has no order position under N in CA, so it should not change the order because the other child nodes of N could not relate to it in CA. In the same way, a node that is deleted or moved from under N doesn't have a position in V, and can't lead to a change of order. The same reasoning applies to property order changed by PAs and PDs. PUs don't change the positions of the properties, so they make no difference. The child nodes or properties in the order representation will only be the ones that belong to the node in both CA and V. These are the elements that partly will be part of the UC of the change, as seen in figure 3.6. They are only partly part of it because we only change the order and only need the IDs or property names for that. The child nodes themselves and the values of the properties are not affected or relied on,[3] which means they're not part of the UC.

If only two out of 10 child nodes to a node change order, we can deem the other 8 nodes as having nothing to do with the change. At the same time there can be other order changes among the other 8 nodes. Would we want to represent those as other changes? Doing so would complicate our algorithm

[3]NRs don't change the paths of the nodes as moves do.



(a) *Reordering nodes. Added, deleted and moved nodes are not part of the UC.*



(b) *Reordering properties. Added and deleted properties are not part of the UC.*

Figure 3.6: *Reorderings. The numbers are the positions that the items had in the CA.*

extensively. We choose not to go into detail about how to analyze exact changes in order that have been made to a number of child nodes or properties because of the enormity of the amount of combinations of changes there can be. They grow exponentially with the number of items in the list. Therefore we choose to represent the order of the nodes or properties by a list of either the IDs of the nodes or the names of the properties in the desired order. This can lead to unnecessary conflicts later, but we decide not to prioritize this part of the thesis.

3.3 Change combinations

This section will analyze and present the combinations of two changes we can have on models, with both changes in one version or one change each in two different versions. Combinations of three or more changes will be discussed later in 3.4, because it will fit better after the analysis of how to manage conflicts and syntax violations. This information will later be used for the change detection and conflict detection parts of the algorithm, found in 4.2.2 and 4.2.3, as well as for the test suite. It is also vital for us to know the inconsistencies that can appear between two versions before we can deal with them in 3.4.

We will first present some prerequisites needed for comparing changes to know how to do it and what we need to think of (3.3.1). In the next section we go through change combinations in one version to see which are possible and when (3.3.2). In the last section combinations of changes in two versions will be analyzed to find when there are conflicts and syntax violations (3.3.3).

3.3.1 Prerequisites

Here we will discuss how two changes can differ from each other and describe this in a structured way. We will then discuss the cases of references and moves in more detail to use in the next two sections.

Attribute differences

We will be looking at combinations of two changes (**C1** and **C2**). Because of the massive amounts of combinations (even though there are only two changes), we have taken inspiration from Grune [6] to present them in tables. To be able to cover all combinations of two changes, we need to look again at the attributes of a change, and see how they can differ.

Version Can be either V1 or V2, and since none of them is prioritized over the other, we have only two choices; either the changes are made in the *same version* or in *different versions*.

Position The changes C1 and C2 are pertaining to certain nodes (**N1** and **N2**) in the model, and these nodes can only be related to each other in three different ways. If we start checking the paths to the changed nodes, going from the root and down, we will see that the paths will either at some point differ from each other or one or both of the paths will end. We thus have three cases, the changes are in *a) different subtrees*, or in the *same subtree*, in which case they can be on *b) different nodes*, with one being

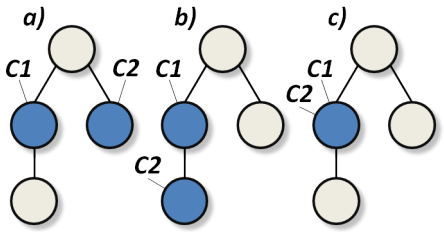


Figure 3.7: *Two changes can be positioned in a) different subtrees, in b) the same subtree but in different nodes or in c) the same nodes.*

located over the other (e.g. N1 over N2), or on the *c) same node*. See figure 3.7.

Type Since there are plenty of different types of changes, this is what makes the analysis extensive. We have eight kinds of changes,[4] and both C1 and C2 can be any one of them, which means 64 combinations for each combination of version and position. We will see, however, that many of them fall away because of duplications and many are very similar.

Name For data changes, the properties changed can have either the *same name* or *different names*. What the names are does not matter to us, just if they are equal or not.

Value The same as for names, we can have the cases where the changes have either the *same value* or *different values*. If the value is a reference we can have other cases, which will be discussed below.

Order The new orders of two reorderings (NR or PR) made to the items of the same node can be either the *same order* or *different orders*.

Order position For changes that lead to moving the same node to the same parent (MDs) or adding the same property (PAs) to the same node, the position in order can vary. They can either be the *same position* or two *different positions*. Differences in order positions are not possible for items changed by other types of changes (except reorderings), because they are on items that already are part of the node in CA, so they will have the same positions.[5]

We will in our analysis go through every combination of these attribute differences, splitting our analysis up into two sections where we look at changes in the same version, and then at changes in different versions. Each of these sections will be split into three parts where the three position combinations are covered, and in those sections every combination of types, names, values, etc. will be analyzed.

If we would have had models where we could not have connections between two different parts of the model, and we could not move our nodes, just using these attributes for checking if they're locally (in CA) changing the same item and in what way would be enough to find every conflict. Unfortunately, this is

[4] We sometimes split up moves into MSs and MDs because they have different locations.
[5] When we take other changes into account, and if no reordering has been done

not the case. Therefore we will discuss references and moves and their impact in the following sections.

References

References play an important role in our analysis since they are the only means by which we can connect two different subtrees of the model. This means that changes creating, changing or moving references can affect parts of the model outside the UC of the change. Changes in different versions that are connected to the same references can therefore lead to syntax violations. As explained when going through the change types in 3.2.3, the only three change types that can lead to new references are NAs, PAs and PUs.

There is a syntax violation that can appear due to the creation of references, and we will call it **reference breaking**, see figure 3.8. If a new reference is created, the node that it refers to needs to exist for the reference to be syntactically correct. Removing the node referred to therefore leads to a syntax violation. We can deduce that only NDs can lead to the non-existence of a node, since moving it doesn't change its ID, just the path. And we can't have a PD or a PU pertaining to the ID property because it identifies the node and can't be changed. Deleting the property with the reference itself deletes the whole reference and there's nothing to worry about when it comes to breaking the syntax. Thus, the only way to get a syntax violation because of a broken reference is to create a new reference in one version and delete the node it points to in the other version.

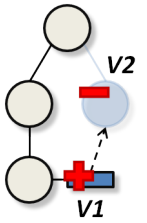


Figure 3.8: *A reference breaking. The node that is referred to in one version is deleted in the other version.*

Moves and positions

We described the three possible position combinations of two changes earlier in figure 3.7. However, we did not consider moves. If every node kept the same position in V as they had in CA, there would be no confusion about where a change actually took place, but since we can have moved subtrees in V, we need to decide on which position to use. This needs to be cleared up before the comparison can begin, and we need to remember this when we are discussing combinations with moves.

If we consider the act of comparing CA with V to find changes we can, when finding a moved subtree, simply compare it with the one in CA as it is, because the only difference the move makes is that the parent of the moved node is different (and the paths to the root from every node in the subtree is different).

When we are comparing a node in CA with its corresponding node in V, we compare their parents to see if the node has been moved. If we then continue to compare the descendants of the moved node, they will still have the same parents, so there won't be any problem. That the paths to the root are changed makes no difference when looking for changes, unless we use the paths instead of the parents for checking if a node has been moved in V. In that case we can compare the moved subtree with the one in CA as if it was still in its original position, keeping the same paths, but we won't be doing that in this thesis since we compare the parents to check for moves.

When we are comparing two changes C1 and C2 in two versions we will go through the different position combinations and check combinations of change types. We can see that in case *c)* in figure 3.7, it doesn't make any difference if or where the changed node has been moved, because the changes are still in the same node. But a case *a)* can turn into a case *b)* and vice versa by moving N2 to or from the subtree of N1. See figure 3.9. These are the only cases where a position combination can change, because they both involve a move change made to a node between N1 and N2 (or to N2 itself), separating or bringing the two changes together. Since we rely on the move change we end up with a combination of three changes instead of two, and we will deal with such combinations in 3.4.3. In 3.3.3 we will therefore not consider these combinations and stipulate that the positions of changes have not been changed through moves, unless the comparison directly involves a move change.

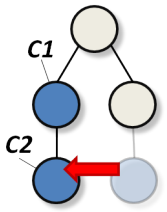


Figure 3.9: *A move can lead to that the position combination of two changes are different in CA and V. Here from different subtrees to same subtree.*

3.3.2 In the same version

Combinations of changes in one version are only analyzed to check which changes are possible, so that we can make sure that we find them all. We won't find any conflicts or syntax violations here, because we rely on that the XMI in the input files is valid.

We'll cover the combinations by going through the three different position combinations one by one. The analysis is presented in a structured way in table 3.1.

Same version, different subtrees

Case *a)* in figure 3.7. See table 3.1a.

If the changes are in different subtrees, then the UCs of the two changes

can never overlap.[6] This leads us to conclude that every single combination of change type, name or value is possible, as long as references are not considered.

We can have combinations where the creation of a new reference in one subtree can lead to the impossibility of removing the node referenced to. These are the combinations of an ND with an NA, PA or PU. This is a quite minor notion, and it won't affect the way we go through the model in the algorithm, because the fact that it's impossible to do some of these combinations doesn't make it impossible to make most of the combinations we're looking for.

Same version, same subtree, different nodes

Case *b)* in figure 3.7. See table 3.1b.

If one of the changes (C1) is made to a node over the other change's (C2) node, we have combinations that are not possible.

If C1 is a data change (**PA**, **PD**, **PU**), its UC will never overlap the UC of C2 because they only change the properties, so all combinations where C1 is a data change are possible. The same goes for when C1 is a **PR**, for the same reason, or an **NR**. As we've said, if an NR is made on a node N it is not possible for the child nodes of N that are added, deleted and moved in the same version to be part of that NR's UC, but that doesn't mean that they can't exist under N.

The case is similar for moves. We can have any type of change under a move. Actually, any change that is below a move will be positioned to it the same way both in CA and in V, which means that when C1 is a move it's considered as one type, while if C2 is a move we have to split it up into **MS** and **MD** (as seen in the table), because they can be in different subtrees. This is a case where the UC of two changes overlap, since we say that the UC of a move includes everything it moves. This is only possible because moves only change the path to nodes, and nothing else, which makes it possible to still identify and make changes to the nodes that have been moved.

If C1 is an **NA** or an **ND**, it's more problematic. We can't have changes made to an added subtree or to a deleted subtree. As we've already said, though, we can have an MD under an NA and an MS under an ND, since they are not part of the UCs of the NA or the ND. This also leads to the fact that we can have any type of change positioned under an NA or an ND, but only in combinations with a move. We have to also keep in mind that the MS and MD of a move can't both exist under the same NA or ND, since the node has to come from somewhere that existed in CA and at the same time go somewhere that exists in V.

Same version, same subtree, same node

Case *c)* in figure 3.7. See table 3.1c.

Since the composite changes **NA** and **ND** include the node in question in their UC, no other change can be made on the same node at the same time. No combinations with them are therefore possible, as seen in the table.

If we consider **moves** in this situation, we will realize that we don't need to worry about the difference between MS and MD since wherever the node has

[6] Presuming that both the MS and MD of a move are in different subtrees from the other change.

C1 \ C2	NA	ND	MOVE	PA	PD	PU	NR	PR
NA	P	P/NP[1]	P	P	P	P	P	P
ND		P	P	P/NP[1]	P	P/NP[1]	P	P
MOVE			P	P	P	P	P	P
PA				P	P	P	P	P
PD					P	P	P	P
PU						P	P	P
NR							P	P
PR								P

(a) *Change combinations in different subtrees.*

C1 \ C2	NA	ND	MS	MD	PA	PD	PU	NR	PR
NA	NP	NP	NP	P[2]/NP[3]	NP	NP	NP	NP	NP
ND	NP	NP	P[2]/NP[3]	NP	NP	NP	NP	NP	NP
MOVE	P	P	P	P	P	P	P	P	P
PA	P	P	P	P	P	P	P	P	P
PD	P	P	P	P	P	P	P	P	P
PU	P	P	P	P	P	P	P	P	P
NR	P[4]	P[4]	P[4]	P[4]	P	P	P	P	P
PR	P	P	P	P	P	P	P	P	P

(b) *Change combinations in the same subtree, but different nodes (C1 is above C2).*

C1 \ C2		NA	ND	MOVE	PA	PD	PU	NR	PR
NA		NP	NP	NP	NP	NP	NP	NP	NP
ND			NP	NP	NP	NP	NP	NP	NP
MOVE				NP	P	P	P	P	P
PA	SP				NP	NP	NP	P	NP
	DP				P	P	P		P
PD	SP					NP	NP	P	NP
	DP					P	P		P
PU	SP						NP	P	P
	DP						P		P
NR								NP	P
PR									NP

(c) *Change combinations in the same node.*

Table 3.1: *Possible combinations of two changes in the same version. The dark gray areas are there to not duplicate combinations. Read 3.3.2 for detailed information.*
[1] *NP if the NA/PA/PU creates a reference to the subtree of the ND.*
[2] *Combined with one of these changes any change on the same row is possible.*
[3] *NP if the move (both MS and MD) is made within the subtree of C1.*
[4] *The node that the NA/ND/MS/MD pertains to isn't part of the NR's UC.*

C1 = change 1
C2 = change 2
NA = Node addition
ND = Node deletion
MS = Move source
MD = Move dest.
PA = Property addition
PD = Property deletion
PU = Property update
NR = Node reordering
PR = Property reord.
SP = same property
DP = different prop.
P = possible
NP = not possible

been moved from/to in the model, it's still the same node (which is why they've been combined in the table). We can, of course, also not move the same node to two different parents in the same version. To do that we would need to have two copies of the moved subtree in different locations, which would mean that the input file would not be syntactically correct. This makes a combination of two moves impossible. We can however have any kind of data change or reordering made on a moved node, for the same reasons as in case *b)*.

Looking at the data changes (**PA**, **PD**, **PU**) we also need to consider the properties that are being changed. We quite obviously can't have two different data changes on the same property in the same version, because they all affect the value of the property, the smallest part of their UC, and it can only be changed in one way from CA to V. Combinations of changes of the same properties are therefore impossible. Any combination of data changes involving different properties are however possible.

NRs can be in combination with any of the data changes since they don't change the same types of elements. For **PR**s we need to consider the types of the data changes, though. As we've said before, properties affected by PAs and PDs can't be part of the UCs of PRs, which means that they can coexist in the same node, as long as they don't change the same properties. PUs, however, only changes the value of the property, and not its name, which is what PRs use to represent a property in its order representation, and the value is not part of the PR's UC. So we can very well both update a property and reorder it. In this case, as opposed to the move case in the previous section, the UCs of the two changes only seem to overlap each other, but in actuality they don't.

3.3.3 In different versions

We will go through all combinations of two changes (C1 and C2) made by two developers (D1 and D2) in two different versions (V1 and V2) to find all kinds of inconsistencies that can appear because of them. In the next section, 3.4, we will discuss how to best take care of these problems. This part of the thesis is a very important part both for the algorithm as well as for the test suite, as we strive to find every inconsistency there is between two versions.

First we will discuss the differences between conflicts, syntax violations and context issues before we start looking for them. After that we'll cover the combinations by going through the three different position combinations one by one. The analysis will be presented in a structured way in tables 3.2 and 3.3.

Inconsistencies

When we compare changes from two different versions we can get inconsistencies of different types, and we need to explain their differences. These can only appear when comparing two versions.

Conflicts appear if there are two changes in different versions made to the same part (node, property, value). The only way two such changes can keep from being in conflict is if the changes are exactly the same change, or if one can be incorporated in the other because of their nature. In the cases of the changes being exactly the same, this can still be seen as a conflict, or at least a context issue, because we can't know what the intentions of

the developers were when doing the exact same change. In our algorithm we strive to represent the information needed to apply both options for the developer, and when the options are the same we don't need to see it as a conflict and simply just apply one of the changes (which one doesn't matter since they are the same). This is because we would like to keep the number of inconsistencies low, as they can lead to more inconsistencies, as we will see in 3.4.2.

Two conflicting changes can't be applied at the same time.

Syntax violations appear when two changes in different versions together violate the XMI syntax. Because we have XMI valid input files, applying one change can't cause a violation, but we must have two changes from different versions that affect the same part of the model while not being in direct conflict. These kinds of connections are only possible through references (which we have discussed in 3.3.1), and *reference breakings* is the only way that we have been able to find that only break the syntax while not leading to a conflict.

Two changes that together create a syntax violation can be applied at the same time, but they break the syntax.

Context issues are the hardest to analyze, because they are combinations of changes that result in logic faults on a higher level, but do not break the syntax of the output file. We can't deduce much of what a developer intended with a change just from the syntax of the model, and there are many changes that may be made in completely different locations of the model, but that still can create context-related problems. We have no way of finding every single of these issues. This is one of the reasons that the developer gets informed about every change that has been made in the different versions, to help him find these issues himself. We do however want to help as much as possible, and will therefore warn the developer when we manage to find probable context issues.

Two changes with possible contextual relation can be applied at the same time and won't break the syntax, but could create logical faults.

These are the kinds of inconsistencies that we will try to find in the following three parts of this section.

Different versions, different subtrees

Case *a)* in figure 3.7. See table 3.2a.

We can again conclude that there are no ways, except through references, that two changes in different subtrees can affect each other. **Reference breakings** can happen if C1 is an NA, PA or PU and creates a new reference and C2 is an ND that removes the node referenced to. The changes are not made to the same part of the model, yet they make the XMI syntax invalid.

As said in the previous section, combinations of two changes in different subtrees can create context issues, but the changes are not close enough to each other for us to be able to find and be sure of these issues. We don't want to warn about things that don't have anything to do with each other, so we don't want to make any assumptions about changes being related contextually here.

Different versions, same subtree, different nodes

Case *b)* in figure 3.7. See table 3.2b.

If C1 is an **NA** it means that there is no possibility for C2 to exist under it because the node and subtree that was added in V1 doesn't exist in CA and therefore neither in V2. Therefore, no change combinations with C1 being an NA is possible.

In the case that C1 is an **ND** we have many conflicts. This is simply because a subtree can't both be removed and changed at the same time. If we under an ND in V1 in V2 either add a node, move a node under, add or update a property or reorder child nodes or properties we have a conflict with the ND. Should C2 instead be a deletion of a node or a property we can see that it can be incorporated into the ND because it does the same thing, only to a smaller part of the model. We decide to not look at this as a conflict because both of the changes can be applied at the same time. This could however be a context issue where D2 wanted to delete a part of the model and keep another, while D1 deleted both of them in one deletion.

If C2 is an MS it has been moved from the subtree and is therefore not deleted. We've said that a moved subtree under an ND is not part of its UC, which is true if both changes are made in the same version, and that is not the case here. Both these changes can be applied without breaking the syntax, so there's no conflict and no syntax violation. They could on the other hand lead to a context issue, where D1 intended to delete the whole subtree and D2's move prevented this.

As we've discussed before, we need to consider the fact that subtrees under the ND can in V1 have been moved somewhere else. If C2 is made to such a subtree, there can not be an inconsistency with the ND. In the cases mentioned above, all C2 changes need to be done in the UC of the ND to create inconsistencies.

When C1 happens to be a **move** we can have any type of change underneath it without creating a conflict, because the move doesn't affect the nodes in the moved subtree. We will have probable context issues in each of these cases, though. Any change made by D2 could be a change intended to stay where it was, and D1 moved it. Just as in the case of NDs, these combinations count only if N2 is in the UC of the move, meaning it isn't moved from the moved subtree. In those cases the change will make a probable context issue with that move instead.

Just as in combinations of changes in one version, if C1 is a data change (**PA**, **PD**, **PU**) or a **PR**, its UC will only be a part of N1 (the node that C1 pertains to) and therefore not affect any node underneath it. This means that it can not be in conflict with any change C2 made to any node under it.

We can, however, have conflicts if C1 is an **NR** and C2 is a structural change to one of the child nodes of N1. The NR's UC includes the child nodes of N1 that exist both in CA and V1, which means that if C2 is an NA or an MD, there won't be any conflict because those nodes are only child nodes of N1 in V2, and not in CA or V1. The nodes will however be very hard to insert in the right position because the nodes have been reordered, and we have no way of knowing where a correct position would be for the nodes, unless we analyze the order more deeply (which we will not do in this thesis). The nodes are therefore added at the end of the list to create a uniformity of how to take care of these

C1 \ C2	NA	ND	MOVE	PA	PD	PU	NR	PR
NA	NC	NC/SV[1]	NC	NC	NC	NC	NC	NC
ND		NC	NC	NC/SV[1]	NC	NC/SV[1]	NC	NC
MOVE			NC	NC	NC	NC	NC	NC
PA				NC	NC	NC	NC	NC
PD					NC	NC	NC	NC
PU						NC	NC	NC
NR							NC	NC
PR								NC

(a) *Change combinations in different subtrees.*

C1 \ C2	NA	ND	MS	MD	PA	PD	PU	NR	PR
NA	NP	NP	NP	NP	NP	NP	NP	NP	NP
ND[3]	C	NC/CI[2]	CI	C	C	NC/CI[2]	C	C	C
MOVE[3]	CI	CI	CI	CI	CI	CI	CI	CI	CI
PA	NC	NC	NC	NC	NC	NC	NC	NC	NC
PD	NC	NC	NC	NC	NC	NC	NC	NC	NC
PU	NC	NC	NC	NC	NC	NC	NC	NC	NC
NR	NC[4]	NC/C[5]	NC/C[5]	NC[4]	NC	NC	NC	NC	NC
PR	NC	NC	NC	NC	NC	NC	NC	NC	NC

(b) *Change combinations in the same subtree, but different nodes (C1 is above C2).*

Table 3.2: *Possible combinations of two changes in different versions. The dark gray areas are there to not duplicate values, and also shows impossible combinations. The darker red areas indicate certain conflicts and the lighter red areas indicate context issues or possible other inconsistencies. Read 3.3.3 for detailed information.*

[1] *Syntax violation only if the NA/PA/PU creates a reference to the subtree of the ND.*
[2] *Improbable context issue between deletions.*
[3] *These combinations take for granted that C2 is in the UC of C1, meaning C2 is not in a subtree e.g. which is moved from under C1 in V1. Such changes never cause any conflicts with C1.*
[4] *If the NA/MD adds a child node to the node the NR is made on, there is no way of being able to position it right in order, so they are put in the end.*
[5] *Conflict only if the ND/MS is made to a child node of the node the NR is made on.*

C1 = change 1
C2 = change 2
NA = Node addition
ND = Node deletion
MS = Move source
MD = Move dest.
PA = Property addition
PD = Property deletion
PU = Property update
NR = Node reordering
PR = Property reord.
NC = no conflict
C = conflict
SV = syntax violation
CI = context issue
NP = not possible

cases.

If C2 is an ND or an MS, those changes will in almost every case be in conflict with a change in V1. In most cases with the NR (C1), because if there hasn't been made any other changes to N2 (the node that C2 pertains to) in V1, it will be part of the UC of the NR. But if N2 is also either deleted or moved in V1, C2 will be in conflict with that change instead, unless N2 happens to be deleted (or moved to the same destination) in both V1 and V2, in which case there is no conflict. We will cover those cases in the next section.

Different versions, same subtree, same node

Case *c)* in figure 3.7. See table 3.3.

If the changes are made to the same node there are a lot more things that need to be checked, because in these cases the properties matter too. We will however see that the change combinations with the structural changes are very alike the ones in case *b)*.

We also have cases where the changes are exactly the same except that they are made in different versions. Because the changes are made on the same items, they can be considered conflicts. As we have discussed in the beginning of this section, however, we don't see these cases as conflicts, because they are not inconsistent with each other and we can apply one of them (which one doesn't matter). We will warn the developer about the fact that there have been made two changes that are the same.

Just as in the previous section, we can't have any changes in combination with an **NA** because the node in question doesn't exist in CA. We can't have two NAs of the same node either, because we rely on an ID system which gives every newly created node a unique ID. If somehow we have the exact same ID of two newly added nodes (say because D1 copied the node directly from D2's workspace), they could be the same change if the added subtrees are exactly the same, but it's probable that they are not. This would open up a whole new area of possible conflicts, which we don't want to have to consider. We therefore stipulate that every node that exist in both V1 and V2 also exist in CA. If a copy such as the one suggested above has been done, this algorithm will (try to) add both subtrees which will break the XMI syntax.

If one of the changes (C1) is an **ND**, we once again have conflicts. If C2 is a PA, PU, NR or PR we can't apply them to the output file together with the ND because the ND's UC covers their UC and we have conflicts in all of those cases. We also have a conflict if C2 is a move, since we're trying to move the actual node that is being deleted and not one of its child nodes or descendants. If the other change is also an ND, they are the same change and we have no conflict, but will inform the developer about this. If it's a PD we don't have a conflict either, but we can have a context issue, like in case *b)*.

In the case of C1 being a **move** we once again have possible context issues because D1 moved a change that D2 made in a certain place. This is in case the change didn't pertain to the node as a whole (like NA and ND, whose cases we've already covered), but part of it, like the data changes and reorderings.

When the same node is moved in both versions we have to look at the destinations of the moves. If the moves are to different new parents, we always have an inconsistency. However, it's hard to say whether it's a conflict or a syntax violation. The same UC is clearly being changed in both versions, but

C1 \ C2		NA	ND	MOVE	PA		PD	PU		NR	PR
					SV	DV		SV	DV		
NA		NP	NP	NP	NP		NP	NP		NP	NP
ND			SC	C	C		NC/CI[1]	C		C	C
MOVE	SD			SC[2]	CI		CI	CI		CI	CI
	DD			C							
PA	SP				SC[2]	C[2]	NP	NP		NC	NP
	DP				NC		NC	NC			NC
PD	SP						SC	C		NC	C
	DP						NC	NC			NC
PU	SP							SC	C	NC	NC
	DP							NC			NC
NR	SO									SC	NC
	DO									C	
PR	SO										SC
	DO										C

Table 3.3: *Possible combinations of two changes in the same node in two different versions. The dark gray areas are there to not duplicate values, and also shows impossible combinations. The darker red areas indicate certain conflicts and the lighter red areas indicate context issues. Read 3.3.3 for detailed information.*

[1] *Improbable context issue between deletions.*

[2] *Possible unprioritized conflict between positions in order of the added items.*

C1 = change 1
C2 = change 2
NA = Node addition
ND = Node deletion
PA = Property addition
PD = Property deletion
PU = Property update
NR = Node reordering
PR = Property reordering
SD = same destination
DD = different destination
SP = same property
DP = different property
SV = same value
DV = different value
SO = same order
DO = different order
NP = not possible
NC = no conflict
C = conflict
CI = context issue
SC = same change

we can still apply both the changes at the same time and not lose any data because they are moved to different locations and thus don't interfere with each other. The problem is that applying both the moves would break the syntax because there would be duplicated nodes with the same IDs. We will consider this a conflict, but considering it as a syntax violation won't change the way we take care of it in 3.4. See figure 3.11a.

If the moves are to the same destination, the changes are not in conflict, except that we also have the order position of the moved nodes to take into account. How these are analyzed is again discussed under *Order management* in the end of 4.2.4. If the child nodes' positions of order are the same, we have no conflict, but if they are not, we have a conflict. We can however deduce that putting the moved node at a certain position in order among its new siblings is most often not of as great importance as actually moving it under its new parent. There is a conflict between the positions, but we will prioritize this conflict lower than other conflicts because the order position is not the main part of the change. We introduce these as lowly prioritized order position conflicts in our analysis. This is not to be confused by the conflicts between moves and NRs.

Let's take a look at the **data change** combinations. We have two data changes that pertain to a certain property each (**P1** and **P2**). We can right away state that if the two changes are made to two *different properties* there can't be a conflict. Thus any combination of PAs, PDs and PUs are possible and never in conflict if P1 $\neq$ P2. The same goes for when C1 is a PR and C2 is a data change, and the PR's UC doesn't include P2.[7]

For the combinations of changes on the *same property* (P1 = P2), we need to analyze a bit more. If C1 is a **PA**, the property doesn't exist in CA, so it can't have been changed in any way in V2, only added. Combining PA with either a PD, PU or PR is therefore impossible, but if C2 is a PA we need to go one step deeper and check the value of the property. If the values are the same, C1 and C2 are the same change (in different versions), if the values are not the same we always have a conflict. In the case of two PAs we also need to (as in the case of two moves of the same node to the same parent) compare the positions in order. In this case we also need to know every PA or PD that's been done to a property of the same node to be able to compare the index values, because they depend on those changes. If the positions are the same, we have no conflict; if the positions are different, we have a lowly prioritized order position conflict, as in the case of MDs to the same parent.

The same reasoning as with PAs can be made in the case of two **PU**s on the same property; same value equals same change (which we will warn about), different values equals conflict. We can also apply both a PU and a PR on the same property because they don't interfere with each other, and thus we can't have a conflict between them.

As for **PD**s, we always have a conflict when a node is deleted in V1 and updated or reordered in V2. But should the property be deleted in both V1 and V2, it's the same change.

Because an **NR** affects the child nodes of a node it can't be in conflict with any data changes or **PR** to the same node, because they affect the properties. This only leaves the cases of comparing an NR with an NR and a PR with a

[7] Meaning they don't affect the same properties.

PR. We compare the items (child nodes or properties) in the order lists for the changes to see if they are the same or different order. If they are the same, the changes are the same; if they are different the changes are in conflict. However, if D1 fixes the order in one way, it might be a natural way to order the items, which means that it's more probable that D2 has changed it in the same way than in some other random way. But since there is only one way to do it the same way and many different ways to do it differently, it's still not that probable that the order changes are the same.

It's also important to point out that the elements in the lists in many cases aren't the same due to deletions and moves in either V1 or V2. For the case where there are deletions and moves in V1, they will end up in conflict with the reordering in V2, as pointed out before. Those items don't have any position in V1, so we will only compare the items that are connected to the node in question in both V1 and V2, and the other nodes will get the positions that they have in the version where they were not deleted or moved, given that they weren't so in both versions.

3.4 Conflict management

We now know the different conflicts, syntax violations and (some) context issues that can appear when comparing two changes, and why and when they can happen.[8] In this section we will analyze how to manage these inconsistencies as well as possible while trying to satisfy our requirements (3.4.1) and what problems the solutions in turn can lead to and how to deal with those (3.4.2). We need this to make sure that all the cases of change combinations are covered. Lastly, we will analyze combinations of more than two changes and how they can be split into combinations of two changes (3.4.3).

This section is very important, because it shows how to deal with conflicts and syntax violations, which is a central issue in merging. It will also show how our analysis covers every combination of changes, in case we deal with the inconsistencies we get in a certain way. Information in this section will be used in later sections dealing with problems when merging and for the test suite.

3.4.1 Resolving inconsistencies

Now, the question is, how do we deal with the conflicts and syntax violations presented in the previous section? As we've said in 2.2, we want to represent them in the output file so that the developer can fix them himself in the best way. We need to supply the developer with all the information needed for this task, so in accordance with our requirement of *no loss of data*, the conflicts and syntax violations will be taken care of in a way that all the info needed for either of the options is present. This will lead to problems with the representation of this information. We will discuss this further in 3.5.2.

When encountering a possible context issue we will, however, apply the changes and only warn the developer about it. This is because we can't be sure that what we find actually is a context issue (we will leave that to be decided by the developer) and because the changes can be applied without breaking the syntax. Thus, we do not need any solution for the cases of possible context

[8] Inconsistencies were described in 3.3.3.

issues, but we will still present them in the following sections, because they can be put in certain groups, and we want to present all the inconsistencies in every group.

The inconsistencies (conflicts, syntax violations and context issues) that we have found can be grouped as follows:

- Node deletion inconsistencies
- Move inconsistencies
- Property conflicts
- Order conflicts

The two top groups include most of the conflicts that include at least one structural change, while the two bottom groups cover the conflicts pertaining to a certain node.

We'll go through these groups and analyze how to deal with them while still protecting our initial requirements in 3.1, that the output should be XMI valid, that no data should be lost and that the output should not depend on the order of the input files. In our solutions we will choose to not apply certain changes, but that doesn't mean that we won't take them into consideration later in the continued comparison between changes. If a change is in conflict with one change, it can still be in conflict with or create a syntax violation together with another change, and we do not want to miss these inconsistencies.

Node deletion inconsistencies

These are the most common and simplest inconsistencies and they appear when a subtree has been deleted in one version and changed in some way in the other version.

These are the changes in V2 that can lead to inconsistencies in combination with an ND of the node N in V1:

- A PA, PU, NR or PR in the ND's UC, an NA or MD in the ND's UC (not including N), a move of N or an NR which N is part of all lead to a *conflict*.
- An NA, PA or PU that creates a new reference that points to N[9] leads to a *syntax violation*.
- An MS in the UC of the ND (not including N) can lead to a *context issue*.
- A PD in N or an ND or PD on a descendant of N can lead to a *context issue*.

The only thing that we need to know to represent an ND is its node and that it's to be deleted, and which nodes under it that have been moved from the subtree. To be able to represent the changes that are in conflict with it we, however, need the deleted subtree. A simple way to give the developer all information needed about both changes is simply to not apply the ND, and put

[9]Or one of N's descendants in the UC of the ND, but this is not possible according to the serialization pattern since only child nodes of the root can be referenced to.

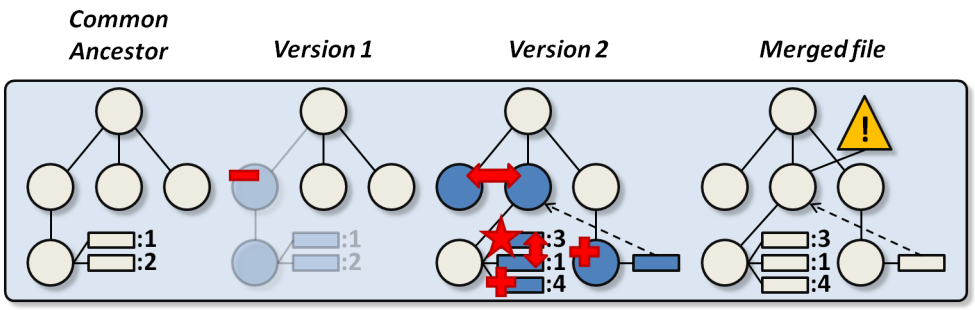


Figure 3.10: *Node deletion inconsisencies. Every change in V2 is in conflict with the ND in V1, except the NA which creates a syntax violation. This figure doesn't cover all the cases. The developer is warned about the discarded ND.*

a warning on the node that was to be deleted saying that it's in conflict with the changes in or references to its subtree. The non-deletion of a subtree can only lead to more information for the developer, and there is no risk for duplicated nodes in the model since every move from the was-to-be-deleted subtree is still applied.[10]

We refrain from applying the ND in each of the cases that lead to a conflict or a syntax violation. See figure 3.10.

Move inconsistencies

Moves mostly lead to probable context issues, but there are cases that are more serious.

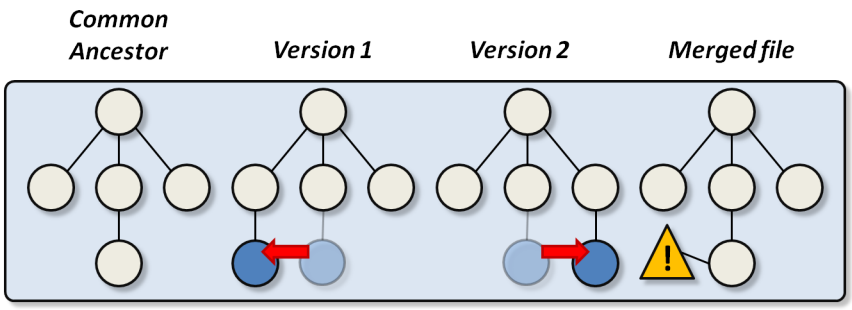
These are the changes in V2 that can lead to inconsistencies in combination with a move of the node N in V1:

- A move of N to a different destination leads to a *conflict*.
- An NR whose UC includes N leads to a *conflict*.
- An NA, ND, MS or MD in the UC of the move (not including N), or a PA, PD, PU, NR or PR in the UC of the move can lead to a *context issue*.

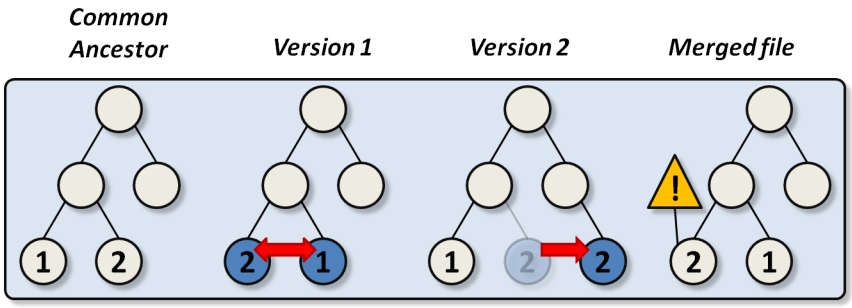
In the first case, we can't apply both of the moves, because it breaks the syntax. There are a few solutions possible. One is to change the IDs of *all* the nodes in one (or both) of the subtrees, which would only lead to more possible conflicts with references, etc. This is not a preferred solution. We could also apply one of the moves and warn the developer that it could have been moved to another destination. However, this goes against our requirement of symmetry, where the applied move would have a much better chance of being used than the other move. The solution settled for is therefore to not apply any of the moves and inform the developer that the subtree was moved to two different positions in the two versions. See figure 3.11a.

The second case makes us choose between moving the node to another parent or reordering it under the same parent. The move seems to be of greater

[10] Given that they themselves are not in conflict with some other changes.



(a) *Conflict between two moves. None of the moves are applied.*



(b) *Conflict between a move and an NR. The move is not applied.*

Figure 3.11: *Move conflicts.*

importance, but we have no way of knowing this. If we want to be consistent, however, we would choose to not apply the move as we didn't apply the ND in a similar conflict before, so that's what we will do, as shown in figure 3.11b.

We refrain from applying the moves in these cases.

Property conflicts

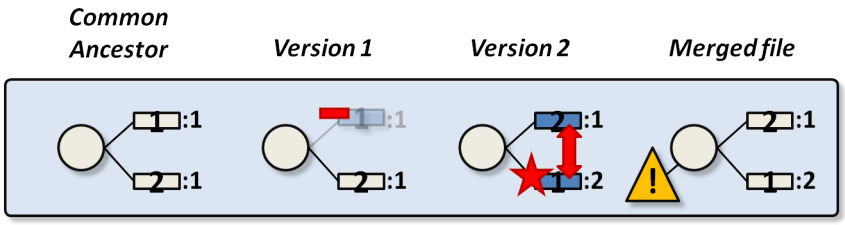
These are the conflicts that appear when we make changes on the same property in both versions. We can't have any syntax violations here, because we need a to combine a data change with a structural change for a new reference to become incorrect.

The combinations of changes on the same property that lead to *conflict*s are the following:

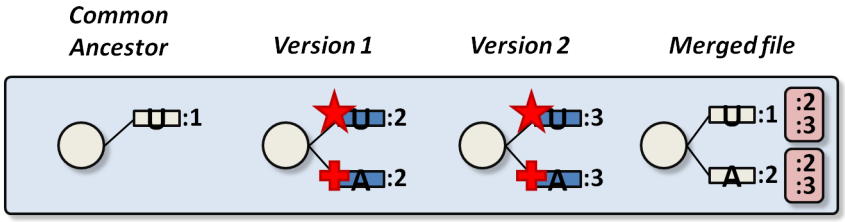
- A PD in one version and a PU or a PR in the other version.
- Two PAs with different values.
- Two PUs with different values.

In the first case we reason the same as we did with conflicts with NDs, applying the PD would lead to losing data, thus we choose to not apply it and apply the PU or PR instead. There will be a warning about the PD that was not applied. *We refrain from applying the PD in this case.* See figure 3.12a.

The second and third cases are a lot harder to deal with. We have two changes of the same type, either PAs or PUs, and both of them have the same



(a) *Property deletion conflicts. Both of the changes in V2 are in conflict with the PD in V1.*



(b) *Property addition and update conflicts. We need to represent the options for the developer in M.*

Figure 3.12: *Property conflicts.*

priority to be applied. This situation is different because we have new values in both changes and thus we need to represent both of them at the same time. Let's deal with the PUs first. In this case the property already exists in CA, so there is an old value that could be used while we somehow represent the options of the new values in a way that doesn't break the XMI syntax, which we will discuss further in 3.5.2. *We refrain from applying any of the PUs in this case, and inform the developer about the choices of new values.* See figure 3.12b.

If there are two PAs the property does not exist in CA and we can't let it keep its old value while not applying any of the changes. In this case we can choose to not apply any of the changes, in which case we have the problem of representing not only the value of the property, but the property itself in our options. Added to this, if we do not insert an added property at its position it leads to that the positions of the later insertions are off. We therefore want to add the property, and when doing so we need to choose a value for it. Since we don't know what kind of property it is, we have no way of knowing which kind of value is needed to not break the syntax, and can therefore not choose a default value, say an empty string. We therefore choose to make an infringement on the symmetry requirement, where we either can choose to apply the change from V1 or the one from V2. *We apply one of the PAs and inform the developer about the other choice.* See figure 3.12b.

Keep in mind that even if these values are in conflict with each other, we still need to find syntax violations they might lead to also. If the values of the PUs or the value that wasn't chosen in the PA conflict would have led to syntax violations because of them being references to deleted subtrees, that doesn't mean that the syntax violations aren't still there, even if they are not explicitly represented. The developer needs to be able to choose one of the options without it creating a syntax violation.

Order conflicts

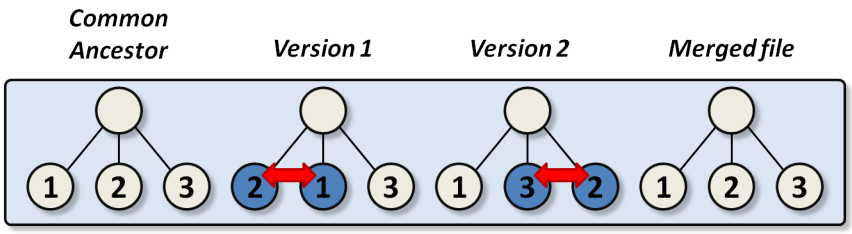
This is the least important group, covering the reordering changes and other changes where only the order is in conflict.

The order *conflicts* include the following combinations of changes:

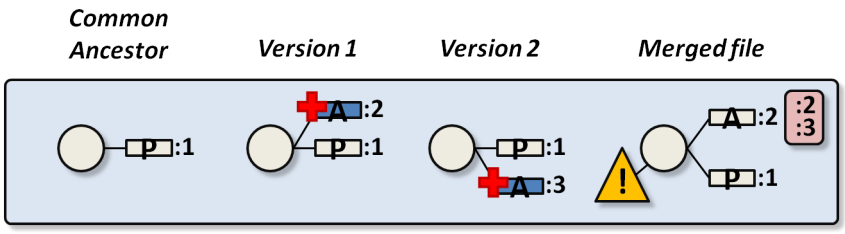
- Two NRs with different order on the same node.
- Two PRs with different order on the same node.
- Two moves of the same node to the same parent, but at different positions in order.
- Two PAs of the same property to the same node, but at different positions in order.

For the two first cases, the simplest and easiest way to deal with them is to just keep the original order and inform the developer that the order of the child nodes or properties were changed in certain ways in the two versions. The developer can then choose a compromise between the orders, and in case there also are added nodes or properties, to see if they have been inserted in a proper position. Refraining from applying the NRs or PRs could never lead to loss of data or a syntax violation. *We refrain from applying any of the reorderings, and keep the old order in this case.* See figure 3.13a.

In the third and fourth cases the changes are in conflict, because they are not wholly the same change. However, the position in order of a new child node or a new property is a lot less important than the actual moving or adding of them. If we were to claim a conflict on this matter and strictly want to stick to our requirement of symmetry, both the changes would be prevented from being applied, which in turn would lead to more possible conflicts or syntax



(a) *Reordering conflict. None of the reorderings are applied and the old order is kept.*



(b) *Order position conflict. One of the positions is chosen, the developer is informed about the other option. Possible other PA conflict is representable.*

Figure 3.13: *Order conflicts.*

violations (in the case of the changes being moves). We instead propose a small infringement on this requirement and apply one of the changes, and inform the developer about the other possible position in order. Choosing one of the positions is also needed to make it possible to represent a conflict between the new values of two PAs of the same property. This conflict is much more needed to be presented to the developer than the conflict between positions in order. Therefore, *we choose to apply one of the two changes and inform the developer that the item was inserted at a different position in the other version.* See figure 3.13b.

3.4.2 Inconsistencies due to resolved inconsistencies

When we fix inconsistencies by not applying certain changes, it can lead to further conflicts and/or syntax violations with other changes. We therefore need to analyze if resolving inconsistencies the way that we presented in the previous part can lead to more inconsistencies.

Let's start by briefly discussing **two changes that are exactly the same**. We chose to apply one of them (they are identical anyway), which in no way could lead to a conflict or syntax violation with other changes in any of the versions, because the change is made in both versions, so they can be applied in combination with every change in both versions. *We have no possible conflicts or syntax violations.*

The conflict that **moving the same node to different parents** led to was resolved by not applying the moves. This means that the old parent of the moved node needs to exist so that it can continue being the parent. The only way a node can become non-existent is if it is deleted in either V1 or V2. It makes no difference if it's moved or reordered. *We have a possible conflict with an ND.*

In the same manner as the previous case, the conflict of **reordering and moving the same node** in the different versions was solved by not applying the move. In this case, however, it's not possible that the old parent was deleted, because the NR pertains to that very node. *We have no possible conflicts or syntax violations.*

The conflict that appears when **updating the same property with different values** is solved by not applying the updates and keeping the old value of the property. The only problem this can lead to is that the old value is an old reference. This could be to a node that has been deleted in either V1 or V2, creating a reference breaking. *We have a possible syntax violation together with an ND.*

Adding the same property with different values leads to a conflict which is taken care of by applying one of the changes. This could never lead to a newly created conflict or syntax violation since no old value is applied. *We have no possible conflicts or syntax violations.*

When **deleting and updating or reordering the same property** we de-

cided to not apply the PD and warn about it. In the case of it being in conflict with a PU, the property gets the value the PU gives it, which can't lead to any new conflicts or syntax violations. Giving the new value removes the value that we wanted to remove with the PD anyway. *We have no possible conflicts or syntax violations.*

In the case of a PD in conflict with a PR, removing the PD leads to that we can have an old reference that points to a node that is deleted in the other version. *We have a possible syntax violation together with an ND.*

When we encounter **conflicts between two reorderings** we simply decide to not apply any of them. Not applying these could never lead to that the model changes in a way that would lead to conflicts or syntax violations. Instead, it actually means that we are able to insert new items at the appropriate places. In fact, if the reorderings are PRs, and the PR in V1 is also in conflict with a PD in V2, not applying the PRs would make it seem like the PD could be applied because the PR isn't in conflict with it anymore. But we won't do that, because D1 still reordered the property D2 deleted for a reason. *We have no possible conflicts or syntax violations.*

The conflicts that appear when **deleting a subtree with a PA, PU, NR or PR in its subtree or a move or reference to its subtree** or an NR that includes the deleted subtree, are taken care of by not applying the ND. If the ND was in V1, there can't have been made any other changes on the UC of the ND in V1. The subtrees moved from under it are not part of its UC and make no difference in this case since those moves will still be applied. The subtree will look exactly the way it does in V2, and the only thing that can be part of the UC of the ND and be connected to any other part of the model is a reference, either added in V2 or deleted in V1, but reappeared. In either case it can't be in conflict with anything except another ND of the node a reference points to. If that happens we could either warn about it or again solve the conflict by not applying the ND. This means that we can have a chain of discarded NDs, but the probability of that happening is minimal, and we choose to do it this way because we want to keep the solutions uniform. *We have possible conflicts with NDs.*

As we can see, we can only get conflicts and syntax violations with NDs, and these are again solved by deciding to not apply the ND. We will use this information in the next section where we analyze combinations of more than two changes.

However, we also need to analyze if it's possible that resolving two different inconsistencies could lead to a new inconsistency. Because of the requirement to not lose any information, resolving inconsistencies in most cases leads to more information, and seldom to less information.[11] We can see that giving more information never could lead to inconsistencies.

In our solutions we have chosen to not apply certain changes. Not applying NDs or PDs leads to more information. Not applying moves only leads to losing the new paths, which can't create an inconsistency with the solution of another

[11] When we give the developer the information through annotations (see 3.5), it's not directly part of the model, and thus in a sense it's less information connected to the model.

inconsistency. When we try to represent the two options in the cases two PAs or two PUs are in conflict some of the values won't be part of the model, but this can't either lead to new inconsistencies.[12] This leads us to conclude that there can be no new inconsistencies due to solving two different inconsistencies. We will use this fact in our discussion in 3.4.3.

Solving inconsistencies can only lead to new inconsistencies in combinations with other changes, and not with the results of solving other inconsistencies.

3.4.3 Combinations of more than two changes

Here we will analyze combinations of more than two changes in the two versions and prove that most of those cases can be split down into cases of two changes and/or a combination of a change and a conflict solution.

As we know, there are only two versions that are being compared, and we need to have at least one change in each of the versions to get inconsistencies. We can have, as seen in the previous section, combinations where two changes create an inconsistency which is resolved and creates a new inconsistency with another change. What we are looking for here, however, are three or more changes that together create an inconsistency where any two of those changes do not already create an inconsistency.

We need two changes that are in the same version to affect the same part of the model. If we look back at our analysis of combinations of two changes in one version in 3.3.2, we can see that in most cases when the UCs of the changes don't cover the same parts of the model they're possible, and if they do cover the same parts they're not possible. The exceptions are references, which affect parts outside of the UC, and moves, which can cover the UCs of other changes with their own UC.

If we for a moment disregard references and moves we can see that without them there is no possibility of having three changes in two versions that affect the same items, and therefore create an inconsistency, because the UCs of two changes in the same version can't overlap. This, of course, is based on the syntax that we have in models, and it doesn't mean that we won't end up with changes that contradict each other on a higher level. It's impossible for us to find all kinds of context issues, so even if a merge goes through without any conflicts or warnings, it doesn't mean that the output looks the way it's supposed to.

Looking at the **references**, they consist of two ends; a property and a node in a different subtree. If we in V1 have a change that creates a new reference we can in V1 also have another change that changes the node that the new reference points to. Remember that the reference is actually just the ID of the node, so the change on the node has to affect its ID to be connected to the reference change in the same version (an update of another property in the node referred to makes no difference). The only types of changes that can affect the ID of a node are NAs and NDs. There is no possibility to have a data change on the ID property, since it identifies the node. Moves don't affect the IDs, they just move the nodes, and since references always point to child nodes of the root, those nodes can't be moved anyway. Obviously, we can't have an ND on the node that

[12] It could if the PUs pertained to the ID of a node, but such changes are not possible.

we refer to in the same version, because then the XMI would be invalid in the input file. This leaves the creation of a new node as the only change that has a direct connection to the creation of a new reference in the same version. Since the node then doesn't exist in CA, there is no possibility of making any changes on it in V2, and we can't make any changes on the parent, because that's the model root. Even if we could, that change would be in conflict with the NA first, and not with the reference and the NA in combination. We therefore can't find any three changes that in combination lead to inconsistencies, even if we also involve references.

Let's then take a look at the impact **moves** have. If we consider the changes which UCs can be affected by moves, it's only composite changes, because they are the only changes that encompass more than one node. If a non-composite change, a change that only pertains to one node, is moved it makes little difference in the comparison between that change and other non-composite changes. As for references, we can't move nodes referred to, and nodes with newly created references can in combination with moves only create context issues.

Moves were discussed in 3.3.1, where we mentioned that a case *a)* can turn into a case *b)* and vice versa by having a move of N2 to or from the subtree of N1, which makes a combination of three changes (C1, C2 and the move). We want to try to find cases where the UCs of these three changes cover the same item, and see what happens in those cases. C1, the change on N1 in V1, has to be a composite change (NA or ND) to affect N2. In the case C1 is an NA, the move has to be to the subtree of N1, and it can only be in made in V1, because the subtree doesn't exist in V2. Because of this, neither the move nor C2 is part of the UC of C1, which we need it to be. If C1 is an ND, C2 can be made in both versions, but always in the same version as the move, because we can't have a change that isn't a move under an ND in the same version. Because we need the UC of the ND to cover C2, N2 has to be located under N1 in CA. Because of this the move has to be made from the subtree and in V2. As for the change type of C2, it has to be a type that lets the changed item(s) exist in both CA and V2, and have the same path to the root (otherwise the move's UC wouldn't incorporate C2). Deletions, additions and moves don't allow this, so the only

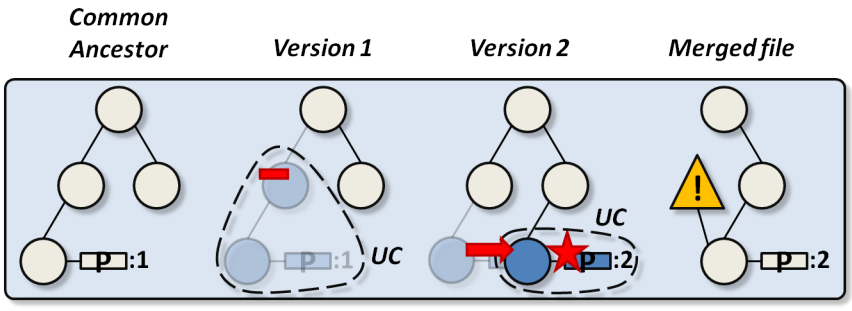


Figure 3.14: *The only case where a part of the model is covered by the UCs of more than two changes. The value of the property P is covered by the UCs of the three changes. There is a probable context issue between the ND and the move, so the user is warned that the moved node was not deleted. The PU could also be an NR or a PR.*

types that C2 can be are NRs,[13] PRs and PUs. These are the only cases where a part of the model is covered by the UCs of more than two changes, and the only thing that we get is a probable context issue. See figure 3.14.

But even though these three changes cover the same parts of the model, we still can see that the ND and the move create this probable context issue by themselves, as discussed in 3.3.3. This does therefore not count as a combination of three changes that creates an inconsistency.

We can only have inconsistencies depending on two changes in different versions or depending on one change and a solution to an inconsistency between two other changes. There is no combination of three or more changes that create just one inconsistency where that inconsistency isn't created by just two of those changes.

3.5 Merging

This section deals with how we want to represent different aspects of the merge result. We will discuss change annotations, warnings and alternatives and how to represent these in the merged file. The part of the algorithm that deals with merging is presented under 4.2.4, where among other things the order of application is discussed.

3.5.1 Annotations

As we said in 3.1, we do not want to lose any data, and we are not able to find every contextual issue that exists. One change can thus be in an indirect conflict with another one without the merge tool being able to find it, as it can only be discovered by the developer. We therefore need to be able inform the developer about all changes made to the model through *annotations*. This information can then be used by the developer when making a decision how to deal with the merge.

We want to make one annotation for each change that is applied. They will have to be connected to the node they are made in since we can't connect them solely to a property, so there could be many annotations on one node. Note that annotations will not be made for changes that are not applied. They will be presented through warnings instead, see 3.5.2.

All the annotations include the *ID of the change*, the information of which *type of change* that has been made, as well as which *version* the change was made in. Any *old values* that exist are also presented. Any other information that needs to be put in the annotations and the positions of them differs between the types of changes that exist. We go through the change types below.

NA Connected to the root of the added subtree, saying it has been added. We do not need to inform about subtrees that have been moved into the subtree, because the moves will be annotated themselves. We don't need more information than this.

[13] Given that the reordered nodes are all in the subtree, and it's not made to the root of the subtree.

ND Since the node is removed, we connect this annotation to its parent and save in the annotation the whole UC of the ND as it was in CA. If any other changes would have been made in the UC in the other version there would be an inconsistency presented instead.

Move This annotation is most easily connected to the root of the moved subtree. The ID of the old and the new parents need to be presented. Connecting an annotation to the old parent could prove impossible if it has been removed, so we choose to not do that.

PA, PD, PU Connected to the property's node. The name of the property as well as the value is presented. In the case of PUs, both the new and the old values are presented.

NR, PR Connected to the node in question. The order in CA as well as the new order is presented.

How to practically apply these annotations is discussed in 3.5.3, but first we will talk about inconsistencies.

3.5.2 Warnings and alternatives

When inconsistencies are found we want to give the developer *warnings*, which are depicted as warning signs in the figures in 3.4.1. We'll discuss what to inform the user about for each of the inconsistencies, first some general attributes and then some more specific ones.

First, the *type of inconsistency* will be shown to the developer, not only if it's a conflict, syntax violation or context issue, but also which of those types. Each inconsistency will be given an ID for it to be identifiable and referred to. There should also be a brief *explanation* of what has caused the inconsistency as well as what has been done to fix it, if anything. Optionally we can have a *grade of its importance and significance.*

Every inconsistency is a combination of two changes, one in each version, and the *IDs of the two changes* need to be part of the warning to connect them. Both, only one or none of the changes may have been applied so that they are present in the merged output file, and the different cases have been discussed thoroughly in 3.4.1. Many inconsistencies of the same type all connected to one certain change can be combined and presented together to save space by putting more than two IDs in the warning, but this is just a presentational optimization and not required.

Let's first cover the **node deletion inconsistencies**. The conflicts and syntax violations due to an ND are resolved by not applying it, and we can therefore connect the warning to the node the ND pertains to. In the cases where resolving inconsistencies leads us to not apply an ND we also do it this way, and the ID of the resolved inconsistency is used. In the cases of the context issues, the ND is applied, and we opt to apply the warning to the parent of the deleted node instead.

As for the **move inconsistencies**, we can connect all of them to the node the move pertains to, whether or not the subtree has been moved. The context issues can be combined. All **property conflicts** and **order conflicts** can be connected to the nodes they pertain to.

3.5.3 Representation approaches

We've now discussed what information we would like to put in the annotations and warnings, but we still don't know how to do this practically in the model. There are at least a couple of options to do this, which we will discuss here.

First off it could be done by using **comments** in the model. There are different kinds of comments that can be inserted, XML comments[14] or special kinds of comments for different model types, such as the comment elements in UML. The advantage of using the latter type of comments is that often they are shown in the editor, but we can't use this if we want to be independent from any model type. As for the XML comments, they are generally not seen in the editor, but only when looking at the code. There is also the possibility that the XML comment tag isn't accepted by the editor the model is opened in . One of the disadvantages with any kind of comment is that they can be confused with other comments. This can be solved by having a certain character sequence in the beginning of the comment to identify it as an annotation for the merge tool.

In XMI there is an element type that can be used for our purposes, which is the **extension element**. The extension element is used for putting additional information which is necessary for different tools into the model, without affecting the model itself. An extension element can have two properties, which are optional; *extender*, which is used for identifying the tool that created the extension so that only our tool will use the information in the extension, and *extenderID* which is an ID used by the extender.

The extension element itself can be put in different locations in the model. It can either be nested in the model, put directly as a child node to the node that the additional information belongs to, or as a child node to the XMI element, outside the actual model, in which case the ID of the node the information belongs to can be put in the extension. We would prefer to use the latter way, both to easier find the extensions and to not clutter up the model. It is important to realize that the "additional information does not need to follow the XMI serialization rules; it can be represented using any legal XML elements" [5] (p. 117). This gives greater freedom of how to represent the information we need.

Using extensions doesn't solve the problem of showing the information to the developer, but it can help out a lot if a tool which shows the annotations in the model is made. This tool could be a stand-alone program, but it would be much more preferable if it was a plug-in to the editor used by the developer so that he can use the editor he is used to when making the changes. This again makes us depend on the editor, but since there is no standardized way in XMI to do what we want, we don't have many options to choose between. A proposed solution to this problem is presented by Bendix et al [4], where a meta-model for merged models is discussed, which would make it possible to represent inconsistencies in a standardized way. This is discussed further in 6.2.

A solution to the problem of representing annotations and warnings is very important to find for this algorithm to work in a good way, or else the developer has to deal with the warnings by looking at them on the XMI level, which in most cases is an environment that is not well known to him. It is however much preferred over using text-based merge tools for models.

[14] Where the syntax is: `<!-- This is a comment -->`.

3.6 Test suite

In this section we will discuss how a test suite that makes use of our previous analysis can be created to cover all the change combinations that can occur. First an overview of the test suite will be given before the test cases are discussed. We will discuss the implementation of this test suite briefly in 5.3.

3.6.1 Overview

What we would like to develop is a test suite that can be used not only for this merge algorithm, but one that can be used as a basis for test suites for merge tools of similar types. Creating a test suite that covers every possible change, and more importantly every possible inconsistency, that can occur when merging XMI files requires using black box testing since it should be usable with different kinds of merging tools. To be able to use white box testing you need access to the code and every white box test suite is developed for a certain program. We can presume that all merging tools will use an original file CA and two versions of that file V1 and V2 and from these three files they will create a merged file M. This is the common ground that the tools have and therefore we need to use these files as our basis to create the test suite. We can let the tool being tested take in the three files and create a result which we compare with a file with the desired output.

There is however a problem that arises when we consider the merged file M. As long as we have changes that aren't in conflict with each other it's fairly simple to decide how to change M, since you only need to apply the changes.[15] But when we encounter inconsistencies there is no standard way of dealing with them or of representing the two alternatives of a conflict in a single file and therefore the merged file will have to be specially created for the tool to be tested. Alternatively, the different kinds of inconsistencies could be flagged in the merged output files to make them usable for the test cases for every test suite. This will then however only test the detection of the inconsistency and not the way it has been dealt with.

The way of dealing with the inconsistencies not only affects the merged output, but also the possible combinations of changes that lead to further inconsistencies, which needs to be taken into consideration. The analysis of changes and combinations of them in this thesis gives a very good ground to do this further analysis.

For merge tools that don't give a batched output, but where the developer needs to deal with the inconsistencies during the merge, the analysis in this thesis can still be of good use for a test suite that tests that all kinds of initial inconsistencies are found and that all the changes are found and made. For every combination of changes that do not lead to any inconsistency of any type these merge tools work like batch merge tools, as they apply all the changes without needing the developer to make any choices. The test cases with such change combinations can be used for testing these kinds of merge tools.

[15] When considering the change itself and leaving out annotations to inform the developer about the changes.

3.6.2 Test cases

In this chapter we have covered the different changes and combinations of changes that can be made to models. These lay out the foundation for the test cases that need to be created for the test suite. We can put the test cases in different groups depending on their nature, as presented below.

- Single changes in one version
- Combinations of changes in one version (table 3.1)
- Combinations of changes in two versions (tables 3.2 and 3.3)
- Change combinations that lead to inconsistencies due to resolving inconsistencies

In these cases it's important to be able to prioritize combinations, as there are a lot of them. For example, the cases where there is an inconsistency are of greater importance than the ones without, and changes in one version in the same subtree are more likely to create trouble than changes in different subtrees. Test cases for making sure that the annotations and warnings are correct are also important.

Other than these there are some other things that could be good to test. For example, when testing to find moves, make sure to test moves made both forwards and backwards in the model.[16] Also, when the last child node of a node is removed, the parent node should be represented on one row and not two, as shown in figure 3.15.

There are also things that can go wrong when thinking of how the algorithm has been implemented. For example, Martini instructed to traverse moved subtrees in the *Model Comparer* after the rest of the model because moves were found by comparing their paths.[17] A critical case to test would then be to see if changes in a subtree moved from a subtree that has already been moved were found, that is, if the moved subtree found when going through the moved subtrees also was checked for changes. As the algorithm has been changed, the need for this test case is gone. This proves to say that some test cases depend on the implementation, and as the implementation can be made in many different ways, we can't think of all of these cases.

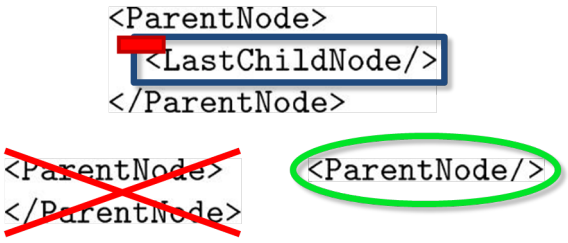


Figure 3.15: *When removing the last child node of a parent node, any remaining text nodes have to be removed too.*

[16] Dealing with both the case when the MS is found first and the case when the MD is found first.

[17] This was discussed in 3.3.1.

Chapter 4

Design

In this chapter we will design the algorithm using the information acquired from the previous chapter. We will present the requirements that we need to cover while doing this. The overview and architecture of the algorithm will be presented before delving deeper into the details of each part of it. These parts will discuss how to find changes and inconsistencies and how to apply these, and how these things are done in the right order for it to work.

4.1 Requirements

This section summarizes what was analyzed in chapter 3, as the result we needed to get from that chapter was all the cases that needs to be covered in terms of finding changes and inconsistencies and how to deal with them. These are the requirements that we need to satisfy when making our design and our implementation.

We still have our initial requirements of having *XMI valid input and output*, having *unique IDs for every node* in the model, having *no loss of data* and having *symmetry*, meaning no higher priority on one of the new versions and that the output files should look the same whichever order the input files are put in. We have, unfortunately, had to break this requirement at times because in certain situations it's impossible to satisfy it,[1] and in other situations satisfying it would make us lose data and break the syntax[2], and we prioritize not doing that.

An important requirement is that we want to *find and represent every single change* that has been made in the two different versions. Which these are is analyzed and discussed in 3.3.2 and shown in table 3.1.

We also have the requirement of *applying the changes* in the output file as they were and at the correct positions (also in order), as long as they are not part of an inconsistency with another change.

The next requirement is to *find all the inconsistencies* that appear between changes in two different versions.[3] These are extensively analyzed and discussed

[1] Like when adding two different items at the same position in the two versions, as will be presented in 4.2.4.

[2] Like when we choose to apply the value of one of two conflicting PAs.

[3] Not all context issues, but the probable ones.

in 3.3.3 as well as 3.4.2, and shown in tables 3.2 and 3.3.

We also have the requirement of *dealing with the inconsistencies* in the way that is presented in 3.4.1, which is important, as our analysis of combinations of multiple changes partly hinges on the way we deal with the inconsistencies.

This leads us to think about the requirement of being able to *represent all the inconsistencies in the merged output file*, as discussed in 3.5.2. This is a hard requirement to satisfy, as what we need to represent is not representable naturally according to the XMI syntax.

4.2 Algorithm

Here we will present an improved version of Martini's algorithm in detail. The whole analysis chapter was a foundation for us to be able to write this section.

4.2.1 Overview and architecture

We have discussed the overview of the algorithm before in 2.3, but we will go into details a bit more here. We need to understand what information we need to be able to go through the different parts of the algorithm.

The architecture of the merge tool is based on the pipes and filters pattern [7], which takes input data and leads it through a filter which changes the data in a certain way and creates the output data. There are three different parts of the algorithm, each one a filter changing the data; the *Model Comparer*, the *Change Comparer* and the *Change Applier*. They are not dependent on each other more than on the output from one part changing the input in the next part. The three parts will be covered in detail in one part each of this section.

Figure 2.4 showed the parallel work between developers D1 and D2 and how their changed models were taken as input into the merge tool and how the merged output looked. In figure 4.1 the actual steps in the merge algorithm are shown for the same input files. This will be used as an example in the discussion below.

We have the two versions of the same XMI file and their CA,[4] and we want the output to be a merged XMI file. What's needed from these files is the actual model part of the XMI, so the files need to be parsed so that we can get model representations that are traversable and easy to get data from. An already existing API for models is preferably used for this. Before we can do this we need to check if the input files are XMI valid. This can be done by using a third party algorithm that is trusted. It is important to remember that the files should be validated depending on the version of XMI that is used for them, and also that the correct serialization pattern is used.

Once these models are obtained, each version paired with the CA will be sent into the *Model Comparer* which will detect all changes made in the version in question. These changes contain almost all the information needed to find every single possible conflict and syntax violation in the next part of the algorithm. The only thing more that is needed is a data structure that keeps track of all the new references in the model. This is the output from the first part. In our example an NA and an ND is found in V1, and in V2 an NA and a PA is found, as well as the reference added by the PA being saved.

[4] We will deal with just one file to make it easier to explain.

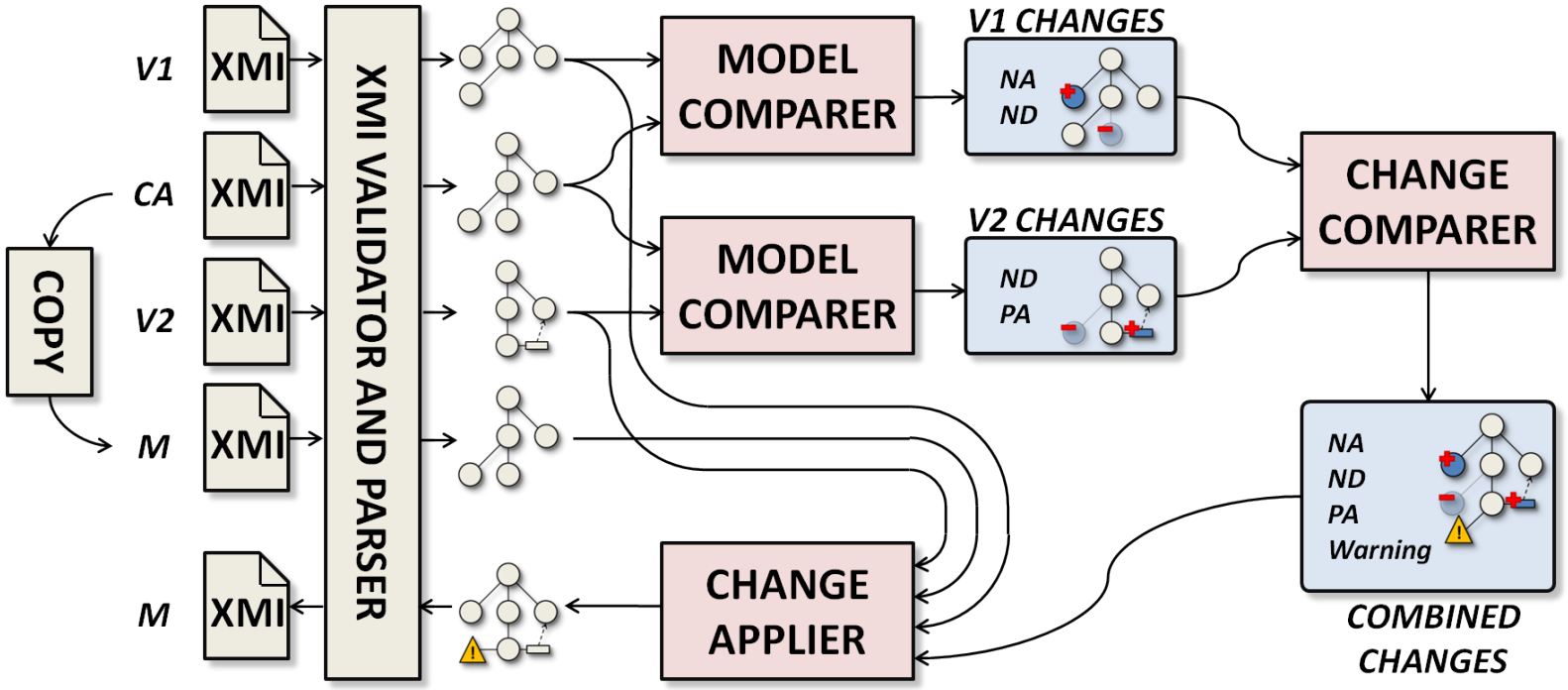


Figure 4.1: *The merge process. The input files CA, V1 and V2 are validated and parsed to create models that are put through the three parts of the algorithm to create the merged output which is serialized back to XMI.*

The two sets of changes and references are the input to the *Change Comparer* which compares them to find inconsistencies. It revises the changes; removes changes to resolve inconsistencies and change them when needed. It also creates warnings and options that need to be represented in the final product. The merged set of changes and the set of warnings and options is the output. Since the ND in V1 and the PA in V2 are in conflict, the ND is marked to not be applied, and instead a warning is added.

Into the last part, the *Change Applier*, is sent the parsed model of a copy of the CA and the combined changes from the Change Comparer. V1 and V2 are also used as input so that items can be put in the right position.[5] The changes and annotations are applied in a certain order to make it right. The output is simply the merged model, which is serialized to the file, creating the merged output file for the whole algorithm.

There are other ways and orders of comparing models and changes and applying changes. An optional approach is to simultaneously go through the two versions and compare them with the CA and as changes are found compare them with how the other version looks and in the merged output file apply the change if possible, and a conflict solution if there's a conflict. For changes that are already applied and later are found to be in conflict with other changes, they can be revised.

The approach used in this thesis has a few advantages to this approach. Firstly, the different parts of the merge algorithm are taken care of in separate parts of the program, instead of all at once, which makes design and implementation of the algorithm easier. Secondly, because of the complexity of the possible inconsistencies it's much preferred to try to find them first after all changes are found, just as it's preferred to apply changes first after all inconsistencies are found, so as to not make any work needlessly (like revising applied changes). Finding the changes that are applied and that possibly are in conflict with a newly found change is also a problem if they're not saved in a data structure where they can be easily accessed, which they are in our approach.

4.2.2 Model comparison

Here we will discuss and decide how we in the *Model Comparer* go through CA and a version V and compare them to find all the changes made in V. We need to make clear which of the models we get what information from, so as to not get confused. For this means every time we refer to an item in CA it will be called CA<item>, and V<item> for an item in V.

The changes that are found are preferably stored in data structures where they can be gone through in the next part depending on their type of change. So we can have an array for every change type. The references should be stored so that we can get every reference to a certain node when we want that, so a map with the node ID as key and a list of the changes that create the references to that node as value works well.

The model consists of nodes which we want to compare with each other. Because of the recursive structure of the model (nodes with properties, having child nodes with properties, etc.) we can create a method that compares two

[5]The reason for this is discussed later in the end of 4.2.4.

corresponding nodes, and then recursively call this method for each of the child nodes that exist in both CA and V. This method will be called the *node comparison method*. We start the algorithm at the root of both the models, CAroot and Vroot, which are easy to find and from which there are paths to the whole model. For the nodes that have been added, deleted or moved we need to take other steps which we will go into below.

This part of the algorithm, as it was presented in Martini [10] (p. 47), didn't go into any details about how to go through the two different models, and is presented in a way suggesting that all the nodes in the models should be analyzed in the same way, despite them being able to exist in one or both of the two models. Looking for new references isn't completely covered there either. And as we've discussed before in 3.2.3, no consideration was given to finding reordering changes either.

We have already validated the files before parsing them and calling the *Model Comparer*. It is however not enough that the models themselves are XMI valid, but we should check and compare the root of the models with each other to make sure that it is actually different versions of the same models that are being used as input. This can be accomplished simply by checking and comparing the ID of the root nodes. If everything is in order, the comparing method can commence, otherwise the algorithm will be stopped.

Comparing properties

The first thing that will be done in the method is to compare the properties of the nodes. This is because these contain the only actual data of the node, and it is a natural place to start. We also want to make these comparisons before we make recursive calls of the method for the child nodes. We could take care of them after the recursive call, but we choose to do it before, because the flow of the program is easier to follow and implement if the recursive call is made in the end of the method.

Now, we need to find the properties that exist in both CA and V to be able to compare them, but we can't be sure that they are in the same order. We can go through each CAproperty from the first to the last and try to find a Vproperty with the same name. If it doesn't exist, we have found a PD, which we store. If it does exist, we compare the values of the properties; if they are different we store a PU, if they are the same nothing has changed. After all the CAproperties have been checked, we can go through the Vproperties that are left and store PAs for each of them. In the cases of PUs and PAs being found we will check if they are new references, as discussed below.

As for finding PRs, this can be done as the CAproperties are gone through. For each Vproperty that is found we can compare and see if the index value is higher than the last one, in which case there has been no change of order. But if it's lower we have found a PR, because the Vproperty was thus found to be located at an earlier position than a property that in CA was located at an earlier position, see figure 4.2. The names of the properties that exist in both CA and V will be put in a list that can be used to create the order lists for the properties should the order of the items differ from each other and we find a PR. The PR can be stored first after all the CAproperties have been gone through.

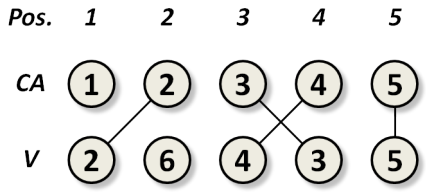


Figure 4.2: *Comparison of the order of properties or child nodes of a node in CA and in V. When two lines cross each other, like those of item 3 and 4, there is a reordering. Items 2, 4, 3 and 5 are part of the reordering.*

Comparing child nodes

After comparing the properties, we will compare the child nodes, which is the only thing left of the node to compare after dealing with the properties. Depending on the outcome we will do recursive calls of the *node comparison method* on them, or something else.

The comparison is done in a similar way as comparing the properties. For each CAchild we try to find a Vchild with the same ID (we need an API that supports this). If it doesn't exist, we have found an ND which will be stored. If it does exist, it can have the same parent, in which case there is no change, or a different parent, in which case we have found a move, which we will store. After going through every CAchild, we go through the remaining Vchildren and check if they exist in CA, but under another parent. If they don't we have found an NA, which we store. If they do exist we have found an MD, which we choose to not store. This is because the MS corresponding to the MD will either already have been found or will be found later (depending on in which direction the move was made), and we decide to store the moves when we encounter their source.

We find NRs in a fashion similar to how we find PRs, except that moved nodes are also part of the nodes that aren't represented in the order.

Traversing the model

The child nodes will be dealt with differently depending on how they fared in the comparison. If there has been *no structural change* made to the Vchild, the node comparison method is simply called recursively. The subtrees that are *moved* in V will be traversed as discussed in 3.3.1, just as if they were not moved, using the node comparison method. The nodes exist in both CA and V, and the change of parent for the node that has been moved is already found. The fact that the subtree is in a different part of the model won't affect the further comparison since we check for moves by comparing the parents of the nodes and not their paths to the root.

For the nodes that are *added* or *deleted* we will call different methods. Since the subtrees of these nodes only exist in one version, the only changes that can be found in them are moves. For every ND we can call another recursive method which traverses the CAnodes and that only tries to find corresponding nodes in V. For every Vnode that exists we have found a move and will store this, and the ID of the moved node will be stored in the NA to show that its subtree is not part of the ND's CA. For these Vnodes we will deal with them as discussed

above, by calling the node comparison method. The NAs are dealt with in a similar manner. Every Vnode in the subtree is traversed using another simple recursive method to find corresponding nodes in CA. However, when a move is found in this case it is not stored because it's an MD and not an MS, and we have chosen to store moves only when we find their MSs. Just as with the NDs, the IDs of the moved nodes are stored in the NA to describe its CA. In the method for NAs we will also look for new references, as discussed below.

As for the traversing order, we choose to go through the model in a depth-first order. It gives the advantage of easier coding with recursive methods. Using depth-first order we can choose to simply make a recursive call of the node comparison method every time we find that CAchild's corresponding node Vchild is not deleted or moved. The downside to this is that for every level we go deeper we still need to keep track of the nodes above, their order and which ones that haven't been traversed. Instead we could go through all the child nodes (but not their child nodes) and find all structural changes and NRs before we go deeper.[6] This way we don't need to store that much information during the traversal.

Finding new references

We have discussed how to traverse the model and find all the changes, but there is one thing left to do that is very important, and that is to find all the new references in the model.

First we will deal with the problem of identifying a reference property when we encounter one. Since we have a requirement to not be dependent on any model type we can't check if the name of the property is one of the names of reference properties for that type of model. We have a couple of choices in this situation. Either we can give the opportunity to the user to provide the names of the reference properties (which could be an error prone solution), or we parse the values of the properties and compare with the format of UUIDs, of which we require the IDs to be. There are also cases where a property can contain multiple references that are simply put in the same string after each other with a space between.[7] We need to take this into consideration when looking for references. The possible problems that can appear due to the possibility of representing multiple references using only one property have not been analyzed very closely.

As we know, only NAs, PAs and PUs can create new references. These can be checked for when going through the models looking for changes. Whenever a PA or PU is found the value can be checked, and in the method for going through added nodes we can go through all the properties to check for references. Remember to not check the ID property, since it's of the correct format but not a reference.

There are also the cases where not applying NDs, PDs or PUs could lead to the reappearance of references that existed in CA but not in any of the changed versions. These can either be checked during the traversal of the model when looking for changes, or looked for later in case they are needed, to not do any work in vain. If saved, these references should not be put in the map with the

[6] Note that this is a combination of depth-first and breadth-first order, but it's still mainly depth-first.

[7] The reference properties `memberEnd` and `clientDependency` in UML2 can have multiple references represented in this way.

found new references, because they are not new references. They can be stored in the changes to which they belong, where they come into use in the next part of the algorithm.

4.2.3 Change comparison

Here we will discuss the comparison of changes in *Change Comparer*. The order in which we decide to go through the different types of changes usually doesn't make any difference, because we won't be removing any changes from the lists. If we find an inconsistency, the change that won't be applied will just be marked as such, so that we can find other inconsistencies depending on that change.

Below we go through the different kinds of inconsistencies and how we go about to find them, in the order they are presented in 3.4.1. The order these are dealt with only matters in one case; we need to find and deal with any conflicts between two NRs and between two PRs before we find any other, because the outcome of this makes a difference when comparing MDs and PAs, as we will see.

When searching for these inconsistencies, for different types of changes we will compare V1 with V2 first, and then V2 with V1. That is, go through the changes of one type in one version first, and then the other. Doing it like this makes it easier to code, but leads to a few cases we need to be cautious about, as follows.

Two changes of the same type made on the same item don't need to be found and dealt with twice, whether they are the exact same change or not.[8] We therefore only need to look for these kinds of combinations by going through the list of changes from one of the versions, otherwise we might get two warnings for the same thing or break the code by fixing something that is already fixed.

For combinations of changes that are on the same item but where the changes are of different types,[9] we will simply only cover them when going through one of the types of changes, and not the other (given that the combinations are possible in the first place).

In the case we have two changes that are identical (except for the version), we not only want to compare them only once, but it's important to make sure that we in the next part of the algorithm only apply this change once, and not try to do it twice. Therefore one of the changes can be marked to not be applied, it doesn't matter from which version, since the changes are the same. It is as usual still kept for further comparison. We will also change the version of the change that is to be applied to show both versions, so that the developer knows that it's made in both versions. This is what will be done for any pair of changes that are identical. Alternatively, we could introduce a new attribute for changes, dealing with identical changes. This attribute would have default value `null`, but once an identical change is found, it could be used to point them at each other.

When we look for inconsistencies below we will do it in a way that all changes that create a certain type of inconsistency with a certain change will be found at the same time so that an inconsistency only needs to be stored at one point and doesn't need to be updated later when a new inconsistency with the same

[8] Like two moves of the same node, or two PUs on the same property.
[9] Like a PU and a PD on the same property.

change is found. Examples of these are multiple changes being in conflict with the same ND, or multiple changes under a moved subtree creating context issues.

Since Martini [10] didn't consider the change of order as a change, his algorithm lacks the consideration of this, both the problem of adding items in their appropriate position and conflicts with reordering changes. This part of the algorithm (p. 48) is also presented in a way that is hard to follow in detail. No comment has been made as to how to deal with changes that are the same, and new inconsistencies that appear through resolving other inconsistencies are only covered a little bit.[10]

Finding node deletion inconsistencies

The lists of NDs are gone through one version at a time, and for each ND on a node N in V1 (vice versa for V2) we look for any NAs, MDs, PAs, PUs, NRs or PRs that have been made in the UC of the ND in V2, if N is part of an NR in V2 and if N itself has been moved.[11] Also, new references to N are looked for, but only in the case that N is a child node of the model root, because references can only be made to those nodes. Any of these cases leads to the ND being marked not to be applied and the creation of a warning mentioning all the conflicting changes, which we store in a data structure containing warnings.

Remember that we need to make sure that we only check the UC of the ND and not the whole subtree of N. Either we have saved the IDs of the nodes moved from under N in V1, or we check for them at this point in the algorithm. It is however easier and faster to use saved IDs, and moves are not that common.

When finding an ND that is to be discarded, we can check for any old references that pop up. Either this has already been done and the references are stored in the change, or we can check the models again.[12] These references can be added to the map of new references. This is because they are now liable to make a syntax violation with other NDs in the same version that the discarded ND existed. This is what will be done for any discarded change that contains old references.

We also look for the cases where there are MSs or other NDs among the descendants of N, or PDs in N's subtree (including N). In each of these cases we create a context issue warning which we store. The NDs and PDs are also marked as not to be applied, since they can be incorporated into the ND above them, and can't be applied after that ND has been applied.

Left is then only the case where there's NDs made on the same nodes in both versions. To find these we only need to go through the list of NDs of one of the versions. In these cases the changes are the same, and they are dealt with as discussed above.

[10] In the case where the old value of two conflicting PUs leads to the old value being used, and if it's a reference to a deleted node, the ND is then discarded.

[11] How to find these changes easily is discussed in 5.2.

[12] To check the models we need to have them as input to the *Change Comparer*.

Finding move inconsistencies

The whole list of moves in one version can be traversed and compared with the moves in the other version to find all the combinations of moves which are made on the same node in the two versions. If we find such a pair and they are moved to different parents we have found a conflict. The moves are marked to not be applied and we create a warning.

If the moves are made to the same parent, we don't have a conflict, and the two changes are marked as the same change as explained above. But we still need to do a comparison of the positions in order that they have. First of all, if there is an NR[13] made to the new parent in either of the versions, the order of the child nodes in the two versions are different and we can therefore not compare the positions. We will therefore not do that in this case, but use the position in the version with the NR. In the case both versions have NRs made to the parent, the position depends on what happened with the comparison between those two changes.[14] If the NRs were the same, then we can continue the comparison, because the order will be the same, just as if there were no NR at all. If they are not the same, however, we can't make any comparison and we end up needing to put the moved node last in the list.

If there is no NR,[15] the child nodes have the same order and we can make a comparison. There are a few ways to make this comparison, and we will discuss those further in 4.2.4. When the positions of the two items have been acquired, and they are the same, then the moves are exactly the same, and nothing more needs to be done, since the changes are already marked as being the same change. If the positions are different, then one of the positions is chosen and a warning is created to inform the developer about the other possible position.

Every move from both versions needs to be gone through to check for NRs to the parents of the MSs in the other version. If such a conflict is found, the move is marked to not be applied and a warning is created and stored.

For every move of a node N in one version we check in the other version for any NAs, NDs, MSs or MDs made in the UC of the move (not including N) and any PAs, PDs, PUs, NRs and PRs made in the UC of the move (including N). If one or many of such changes are found, a warning connected to N is made informing the developer about all the possible context issues found.

Finding property conflicts

We go through all the PDs in one version at a time and look for PUs and PRs made to the same properties in the other version. For each such occasion the PD is marked to not be applied and a warning about this is made. Remember that there can be made both PUs and PRs to the same property, and there's only need for one warning even if a PD is in conflict with both a PU and a PR. Important is also to not forget to check for two PDs of the same property to mark them as the same change.

[13] Which the moved node can't be part of, so there is no conflict in any of the versions.
[14] This is the reason those conflicts had to be dealt with first.
[15] Or if there are two NRs and their orders are the same.

To find all the cases of PAs and PUs in different versions made to the same properties we again only need to go through one list. Every time two PUs of the same property is found, the values are compared and the results (same or different) lead us to either mark both changes as not to be applied or mark them as the same change. In the case they're different and the old value is an old reference it needs to be added to the map of new references, as done with references in discarded NDs. For two PAs with the same name we deal with it the same way as with PUs, except for the old reference part.

There is also the fact that we just as with moves of the same node to the same parent need to compare the positions. This is done in a similar way as done for the moves, again with the need to check for reorderings (one or two PRs) on the node the PA is in.

Finding order conflicts

The only conflicts left to discuss how to find are the ones between two NRs or two PRs. These are simply found by going through the list of NRs and PRs in one version to find a change of the same kind in the other version pertaining to the same node. In these cases both the changes are marked as not to be applied and a warning is created.

Remember that we need to find and deal with these conflicts before we deal with the other inconsistencies, because of the possibility of the result being of need when deciding the position of an item that is added to a node by either two MDs or two PAs.

4.2.4 Change application

Here we will discuss how to best apply the changes that we need to do on **M**, the copy of the CA. Unlike *Change Comparer* where we can look for different kinds of inconsistencies in any order we like, we need to apply some of the changes in a certain order, to make sure that all changes are appliable. We also need to deal with the problem of adding new items into lists in the correct position, and validation of the merged output.

Change type application order

What we start out with in this part of the algorithm is M and all the changes, options and warnings that need to be applied to M. Some of the change types need to be applied in a certain order, but not all of them. The structural changes and the data changes don't depend on each other at all during application, since no structural change removes any node that is used when applying the data changes. Within the two groups there is a preferred order, which is presented and explained below. This is the order we have chosen:

NRs We will start by applying all the NRs, because if a new node is to be added as a sibling to the reordered nodes, its position is in the new order.

NAs Next we apply the NAs. This is because we can't apply moves to added subtrees before those subtrees actually exist in the model.

Moves Once all the subtrees are added we can apply all the moves.

NDs We can only apply the NDs after we have applied the moves since we don't want to delete any subtrees that should be moved from the subtrees that are deleted.

PRs The PRs need to be applied before the other data changes of the same reasons NRs need to be applied before the other structural changes. PRs can however be applied before all the structural changes too.

PAs PAs need to be applied before PDs to make it easier to position items in the right order.

PUs These can be applied whenever, as long as it's after the PRs.

PDs These can be applied whenever, as long as it's after the PRs and PAs.

The changes within a certain change type can be applied in whichever order we would like; top-down, bottom-up, any random order.[16] Contrary to this, Martini [10] suggested applying the moves and NDs simultaneously in a bottom-up order. According to his text we should "apply first the changes in the lower nodes, so that their paths are not changed by higher level node moves or deletions", because otherwise "we are not able to reach [them] by [their] path[s]" (p. 40). This is not needed, however. We don't need to preserve the path to a certain node to access it, since we can easily access it by its ID. To be able to apply any change we only need to make sure that the node exists.[17] This of course means that we need to use an API which supports finding nodes by their IDs, but we have to have this in *Model Comparer* when comparing nodes anyway, so we take this for granted.

For each of these change types we choose to apply V1's changes first and then V2's. This will in most cases not break our requirement of symmetry, since applying one change usually isn't affected by having another change applied before or after it. There are however some cases where changes are affected by earlier changes, and we can't satisfy our requirement of symmetry. That is when we add different items at the same position in different versions (combinations of NAs and MDs or of PAs). In these cases we need to choose to put one item before the other, because they can't be put in the same position. We have a couple of choices in how to do this, as follows.

First, we can choose to just apply the changes in the order they are gone through. Since we've chosen to apply the changes from V1 before the changes from V2, the V2 items will then be positioned before the V1 items.[18] If there should be many added items in a row, all the items from V2 is put before the ones in V1.

Secondly, we can choose to apply the items in alphabetic order, comparing the names or hash values of the items. This way the requirement of symmetry will not be broken, as the items will be put in the same order independent on the order of the input to the algorithm.

[16] Unless the second method under the section *Order management* below is used.

[17] And/or the new parent if the change is a move or an NA.

[18] Since the V1 items are pushed forward to higher index values by the insertion of the V2 items.

Applying changes

How to actually apply the changes to the model and how hard that is depends mostly on the API that is used and what possibilities that are given in terms of accessing and changing the model. Most of the changes that need to be done are not difficult in any way, like putting a new value on a property or removing a subtree. How to do this will not be discussed further in this thesis.

A part of the actual application of changes, however, is the application of the annotations that inform the developer about the change. Since we in the annotations want to be able to represent the old values, the ones before the changes, the easiest way to apply them is while applying the changes themselves because we have access to the old values at this time. The information that should be given in the annotations is discussed in 3.5.1.

Warnings can, unlike annotations, be applied after all the changes have been applied. This is because the information needed for them should already be acquired and saved at the time the warning was created.

When applying NAs we also need to think about not copying the whole subtree from V to M, but only the UC of the NA, which doesn't include subtrees that are later moved to the added subtree. If we don't do this we might get duplicated nodes in the model, which isn't allowed, and depending on the API that is used, this would lead to different things happening, like the new duplicated node not being applied, or the old one being moved automatically, or allowing the model to have duplicated nodes. We do not need to worry about this when applying NDs since the moves have already been applied before them.

We also need to take the order positions into account, which will be discussed below.

Order management

There is still the problem of how to find where to put *new items*[19] in a list when their positions depend on other changes. We can handle this in a few different ways, which we will discuss here.

The first method is the simplest way, which is just to not care about the positions and put every new item at the end of the list, in which case we lose the positions and the items are put in the order they are dealt with, breaking our non-loss of data requirement and also our symmetry requirement. If the order of items is not a high priority when implementing the merge tool, we can choose to implement this solution first and deal with more important issues before taking care of this issue. *This method is not actually a solution to the problem, which we will try to deal with in the following methods.*

The second method is to make use of the index values of the changes made to the list. If we have an addition of an item in V1 and no other changes are made in either version, the item is inserted at the index value that it had in V1. Every deletion (or MS) at a lower index in V2 will lead to that the index value of the addition in V1 becomes lower, and every addition (or MD) at a lower

[19] We will henceforth use this term to refer to items (nodes or properties) added to a list through NAs, PAs or MDs.

index in V2 leads to it becoming higher. We can check all the changes to the same list and figure out the new index value of the new item like this.

However, it's not just that simple. All changes are not made at the same time, and for every change the index values of the siblings change, so we have to decide in which order the changes are to be applied for us to be able to know the correct index values for them to be put in. We need the right position at the right time. We will have to decide the order of applying the different types of changes, which version's changes to apply first and if we should apply changes on items with lower index values or higher index values first.[20] If this method is used, the index values should be calculated after all inconsistencies are found and before changes are being applied, between *Change Comparer* and *Change Applier*. This is because we can get discarded deletions and moves, and MDs and PAs not put at a certain position, which makes a difference for the index values of later items.

Using this method leads to a very strict order of application, and complicated calculating methods, but it works. That is, until you consider moves. A move should be made in one step. Removing a node from its parent first and then adding it to its new parent later after some other changes have been applied is complicated. This is because the subtree that is being moved will not be part of the model and therefore not accessible during the time the subtree is removed from its old parent but not yet added to its new one. We can make sure that we don't access the subtree to apply any other moves before the subtree is added again by applying all the moves in a top-down order to fix this problem. If we consider the other way around where a moved node is added to its new parent before it's been removed from its old parent we also have a problem. We can't have duplicates of nodes in the model since we rely on their IDs to access the nodes, so this way is not recommended. If we were to add a subtree to a new parent we must remove it from its old one before and thus we would need to replace the old subtree with a new node that is there just to not change the index values of its siblings. When the algorithm later reaches this node, it will simply be removed. If we were to use not only the ID of a node to access it, but also its path, the identification of a node wouldn't solely rely on the ID of it but also its parents, in which case we could distinguish between two nodes that actually are the same. But using an approach that relies on the paths to the nodes that we access would lead to other problems, as we've discussed above. Additionally, most APIs would not allow duplicated nodes in the model at any time, since it breaks the syntax.

As we can see, this method is not a simple one and there are a lot of things that need to be taken into consideration, and if there is something done wrong somewhere it can ruin later change applications. *We will therefore discard this method in favor for the next one.*

The third method lets us take care of the problem without using the index values, but we will have to revisit the models of the two versions, and not just use the copied CA model, to be able to make it work. This is the reason that V1 and V2 are shown as input to *Change Applier* in figure 4.1.

A new item is put in a certain position in order among the siblings in the same version, and it's only in regards to these items that the new item has a

[20] This is in case both the change types and the versions are the same.

position. As we discussed in 3.2.3, we can't rely on the previous or next item, because they could be deleted or moved in the other version. But we can rely on *all* the items in the list in the same version, and should all of these be deleted or moved, then the item doesn't have a specific position in the list. We therefore need to know which of the items in the same version that are in the list, and not deleted or moved in the other version, or for that matter, not yet added.

Every single time we add a new item we need to compare the items that are in M with the ones in the version the item was added, and add the item at the appropriate location. This way we don't rely on having to calculate positions beforehand, and we don't have the same problem with the moves. We also make sure that when adding many new items at the same position, like when adding a lot of properties to a node without properties, they will end up in the correct order. This is always true, no matter which order they are applied.

Validation of merged output

The very last thing that needs to be done as part of the merge is to validate the merged output to make sure that it follows the XMI specification. Unless something is wrong with the merge tool, or some case is not covered, this step should always go through without any problems.

Chapter 5

Implementation

This chapter will describe the implementation of the algorithm and the test suite. We will discuss the choice of the API used when implementing the merge tool. We will also bring up the problems that appeared during implementation and how much that actually was finished of the algorithm, as well as the implementation of the test suite.

5.1 Choosing API

The algorithm was decided to be implemented in Java, but an API for the parsing and manipulation of the models also needed to be chosen. There were a few alternatives, although some of them either were not extensive enough, didn't exist anymore (like XMI Framework and JOB) or were not for free (like Rational Rose and Rational Software Architect). The choice in the end was between DOM (Document Object Model) and EMF (Eclipse Modeling Framework).

DOM defines a standard way for accessing and manipulating XML documents, which means that it's not explicitly created for manipulating XMI documents, and thus we would need to be more careful to not break any syntax specific for XMI and not just XML as a whole. DOM provides an easy way to parse and traverse the model and get access to and change its content. What it doesn't provide, however, is a way to access the nodes in the model by using their ID.

EMF provides an extensive framework for creating models which can be used to generate Java code. There are many different parts of EMF which concentrate on different things, e.g. EMF Compare which is EMF's own tool for matching and differenting models. Because of the size of EMF, it is very complicated, and it's not exactly meant for traversing and accessing data in arbitrary models, but to work with specific models in Eclipse. However, EMF supports XMI directly, and most importantly, it has the possibility to access nodes in the model by using their IDs. These are the reasons that we chose EMF as the API, even if it took a long time to learn and understand.

The things that were most irritating when working with EMF was the trouble with traversing and accessing data in the model. To simplify, each node in the model was represented by an AnyType object which had two FeatureMaps, one for the child nodes and one for the attributes. Getting an element out of the

FeatureMaps based on the name of the element was not possible, but had to be done by the index value, and getting the value of an element taken out of the FeatureMap was not possible either, but had to be done at a higher level. This could be worked around by going through the FeatureMaps and putting the data in our own data type for easier access later.

5.2 Algorithm implementation

In the first stages of this thesis the idea was to implement and test Martini's algorithm [10] to show that it could be implemented and what kinds of problems that would appear. The overview of that algorithm was presented in 2.3. As the implementation went on, things that were not taken into consideration in the algorithm appeared, most prominently order changes and the need for keeping the order, and the differences between NAs and PAs, and NDs and PDs. Some changes were made in the design and incorporated in the implementation.

After most of the implementation was done, the analysis for finding all the change combinations was started to create a foundation for the test suite to be implemented. As that analysis went on, more things were discovered, such as the way of looking for a move by comparing parents rather than paths (which changed the way of traversal in the *Model Comparer*), how to compare the order of items, and which inconsistencies that can lead to further inconsistencies, etc. As time didn't allow for further changes in the implementation (or for a full implementation in the first place), the finished product was not implemented exactly the way it has been presented in the analysis and design chapters.

One of the biggest challenges during implementation was to try to find a way to put annotations and warnings in the model and to represent parts of the model in them without them being part of the model itself. This proved to be too difficult and time consuming, and thus it was not implemented in the end. EMF does not at the moment directly support the usage of extensions, which made it a lot harder.

Another hard challenge was to make sure to keep the position in order of items added. This was not prioritized highly, and due to time restrictions a proper solution was not implemented and new items were added at the end of the lists instead.

During the implementation, it was necessary to be able to easily find all the changes in the subtree of a certain node.[1] To make this easier, a `ChangeTreeNode` class was implemented, which could be used to construct a model which is structurally a copy of the model, a change tree, but that only includes the root paths to the nodes that had changes made to them. Every `ChangeTreeNode` can point to multiple changes in the lists of changes that are stored. This way, the subtree of a `ChangeTreeNode` include references to all the changes that exist in the subtree of a change pointed to in that `ChangeTreeNode`. This meant that there was no need to go through every change to check their paths when looking for changes in a subtree.

While the *Change Applier* that was implemented couldn't represent the annotations and warnings, the *Model Comparer* and the *Change Comparer* worked well, and merged output which followed the rules stated in Martini's thesis could be created, although without annotations.

[1] When checking for ND and move inconsistencies.

5.3 Test suite implementation

The test suite implementation that was made was not complete as the implementation of the algorithm and the analysis of the test suite was prioritized higher. The implementation involved the tedious work of creating the CAs, V1s, V2s and Ms for every test case, which was done by using UML models.

Because of the amount of the test cases and the lack of time, only some of them were created. These included around 20 cases of finding and applying changes and combinations of changes in one version, as they are the first and easiest cases to deal with. A handful test cases for inconsistencies, such as node deletion conflicts, move conflicts and property conflicts, were also implemented. In all of these cases no annotations or warnings were applied.

During the implementation, white box testing for the different parts of the algorithm were made, which tested them part by part. This made it easier to know if the different parts worked as they should by themselves. As not the whole algorithm was implemented, not all parts of it could be tested by black box tests. And this did not only apply to the parts that were not implemented (such as applying new items in the right order), but since the annotations for changes and warnings for inconsistencies were not implemented, the proper target models for the merged output could not be created satisfactory to test what was needed in all cases.

For example, if a test for finding conflicts with an ND was made, and multiple changes are put in the UC of the ND, then the merged output file should not have the ND applied. But this does not mean that all the conflicts between the ND and the other changes were made. The changes could be made one by one in different test cases, but then the combining of inconsistencies of the same type into one inconsistency for easier representation in the merged output file would not be tested.

The final test suite implementation that was made covered all the basic cases of finding changes and applying them, but it did not cover enough combinations of changes to say much about how well covering the implementation of the algorithm was. Most of the combinations that lead to inconsistencies were not implemented, but the test cases that were implemented all passed.

Chapter 6

Discussion

In this chapter we will discuss the work made in this thesis. We will summarize the results that have been achieved throughout the thesis, and which of the goals that were accomplished and which were not. The limitations of the algorithm will be discussed, as well as what use the analysis of this thesis can be of for further research and for other merge tools.

Some work of interest made by other people, which are related to that of this thesis, will then be discussed. Lastly, we will summarize and present further research that can be made in this area.

6.1 Results

The algorithm presented in this thesis is a state-based 3-way batch merge algorithm for models serialized in XMI. It takes two versions of the same model and their common ancestor and compares them on the XMI level to find changes made in the two versions and compare these versions to find any inconsistencies of any kind between them. All changes are applied on a copy of the common ancestor, which becomes the merged output. Also any inconsistencies are represented in the merged output.

The goal with the thesis was to use Martini's thesis [10] as ground and further analyze and implement his algorithm to find out if it can be implemented, if it is correct and how well-covering it is. If any of these areas were not up to par, they were to be fixed through further analysis and design.

The implementation of Martini's algorithm was done as part of this thesis, and although it was not completed, it was possible to implement without any bigger problems, except the one mentioned below. Martini's algorithm showed to be work in the cases it covered, but there were many cases that were not covered. These were found and taken care of through further analysis.

We have made a thorough analysis of what differences there can be between changes and what types of changes that can be made to a model. The things that can differ are the version, the position, the change type, the property name and value, and some other minor things. The change types are additions and deletions of both nodes and properties, moves of nodes, updates of values of properties and reorderings of the child nodes or properties of a node.

To make sure that the algorithm was well covering, every single combination

of changes in one version of the model were gone through to find all the changes, and every single combination of changes in both versions of the model were gone through to see if they were creating any types of inconsistencies together, and decisions were made as how to deal with those inconsistencies. Every conflict and syntax violation possible has been mentioned and dealt with in this analysis. We did also find probable context issues, which can be further researched to find more on higher levels for specific model types, but no merge tool will probably ever be able to catch them all.

The restrictions of the algorithm are not that many. The files that are used as input have to be XMI valid and versions of the same model, all nodes in the model have to have an ID unique for the model and the models have to be serialized in a way that no nested entities are possible.

The main problem that lies ahead is that of presenting the information about the inconsistencies to the developer, and representing it in the file without breaking the XMI syntax and without interfering with the rest of the model. Given the importance of being able to merge models, and the complexity of doing so, a standard for representing alternatives in XMI could be created by OMG, but at the moment no such functionality is planned to be introduced.

The analysis and design that has been presented in this thesis can be of good use as reference for further work on model merge tools and test suites for model merge tools of any kind, as the type of algorithm used can differ greatly, but the types of changes and change combinations still are the same on the XMI level.

6.2 Related work

As mentioned many times earlier, this thesis is largely based on Martini's master's thesis [10], which is summarized in 2.3. The algorithm that Martini developed was well analyzed and presented, but not completely covering all cases. Since it is the foundation for this thesis, the algorithms look very similar with some additions and changes in the algorithm presented here. Those changes have been discussed in more detail in 5.2 and 6.1.

In Bendix et al. [3] the jump from using traditional development to using model-driven development (MDD) and the problems it brings are discussed, as well as the experiences of making this jump in Ericsson. Different ways of working with models are described, where the model centric development is being used at Ericsson, where the code is generated from the models and not the other way around.

The experiences of using MDD in Ericsson were positive, as it made it easier for project participants to understand the system and to handle complexity, etc. But the use of version control with models proved to be a problem, as we know. The teams would either have to choose programming instead of modelling, or deal with the version control problem in another way. Solutions have been to make excessive planning in order to serialize work to prevent conflicts, to make manual merges with the model editor, or even to use text-based merge tools and later make manual edits of the models in the file representation of the model, at the XMI level. The algorithm presented in this thesis could be of great help in such a situation, even in its current state without being able to represent annotations and warnings in the models.

The paper brings up many problems that have been encountered, and wishes for the future from the developers' point of view.

The algorithm in this thesis uses a state-based approach, and the other approach to model merge is the operation-based approach. In the operation-based approach [8], the algorithm relies on the editor to record the operations that have been made on the files so that the sequences of operations from the different versions can be merged into one sequence of operations which then later is used to create the merged output file.

Having access to the change operations that have been made to the models and not just the final states of the models, there is surely more that can be deduces as to what intentions the developers had with certain changes. If we for this thesis would have had the actual recorded changes instead of finished state changes, the change combination analysis would have been different, but there are still many things that would be similar. As such, the results that are reached in this thesis could be of use for developers of operation-based merge algorithms.

In Barrett et al. [2] the authors try to remove one of the major drawbacks that comes with operation-based merging, which is the tight coupling between the tool performing the merge and the recorder (editor) tracking the changes made to the models. This is done in three steps: (1) abstracting away differences between change recorder outputs; (2) reconstructing modified models from the change logs; and (3) incorporating change history into the models themselves.

In their paper the authors present how they have incorporated this idea into their model merge tool Mirador. They have created a change record structure in which any type of change record objects (changes recorded from different change recorders) should be able to be represented. The structure contains interfaces so that recorder-specific classes can be implemented to deal with the way that records are represented for certain change recorders. Though this means that there needs to be implementation made for every type of recorder format, it's still better than having to deal with all of them in different ways. This is a step towards making a standard for change recording.

The authors have also created a system in Mirador which gives the user the opportunity to choose how to match the elements in the models, where many different matching strategies can be used at the same time creating an overall score computed with a weighted distance function. The strategies – which may be externally supplied – and their weights are chosen by the user at start-up. In this thesis we have fully relied on the IDs of the elements for matching, which is a limitation. A feature such as this could be of use for the merge algorithm presented in this thesis too, if it's reliable enough.

If state-based and operation-based merging is compared, it's not as much two completely different approaches as it is a simpler and a more complicated approach. In the state-based approach the operations are not available, but in the operation-based approach the states are always available, meaning that there is more information available. The questions to be asked when choosing between the two approaches should be if the state-based approach is enough, and if it's worth the extra space and programming and computational effort to follow an operation-based approach. With the operation-based approach there is also the risk of being dependent on specific tools, which the paper in question tries to erase.

The main thing that separates this algorithm from other algorithms is the fact that it is a batch merge algorithm. In Bendix et al [4], the different problems that batch merge tools face are presented and discussed. The paper focuses on what can be done for creating a standard for dealing with these problems.

The creation of a meta-model that would allow representation of any type of extra information needed for merge tools (alternatives, warnings and annotations) to be possible is discussed. This meta-model would be used for the merged output, and then "the subsequent conflict management phase can be used to gradually bring the merge result in a state that it conforms to the original meta-model" (p. 4), which can be done with a conflict management tool or editor that uses the special merge meta-model. With a new meta-model, XMI syntax violations can be present as long as the syntax of the new meta-model is not broken. This would help make sure that any merged output that contains inconsistencies of the type that breaks the XMI syntax are not put in the repository, since they need to first conform to the XMI meta-model.

Other issues that are discussed are change and conflict mark-up, alternative representation and rich alternative proposal, which are related to annotations and warnings in this thesis. Violation handling is also discussed, as it also has been in 3.4. The proposed solutions for the issues all make use of the proposed merge meta-model, and would then possibly be handled better than they have been in this thesis. However, how this meta-model would look like in more detail is said to be a wide open question.

6.3 Future work

There are plenty of things that aren't totally covered in this thesis, into which more work can be put.

As discussed when introducing reordering changes, it was decided to in the UC of the NRs and PRs include all the items that are part of the node in both the CA and in V. To not get unnecessary conflicts we would like to not cover items that aren't actually reordered. For this we need a **deeper order change analysis**, to be able to recognize exactly which items it is that are reordered, which is a very complex problem. Again, we don't know actually how common reorder changes are, and it might not be worth putting too much effort into this area.

When we started discussing the merge algorithm in 4.2.1 we said that we would deal with just one file to make it easier to explain. If the model is spread over **multiple files**, the algorithm should work, assuming that we actually take that into consideration and parse all the files needed to create the model. If there are any connections between models, that's a harder problem to deal with.

We have taken a certain stance in this thesis when it comes to what we consider **possible context issues**. There are some cases that might be considered context issues that we have chosen not to. If two different properties in the same node are changed in two different versions, that might be significant, but we can't be sure about that at all. Changing two different child nodes of the

same child might also have some significance, but that could also depend on the depth from the model root those changes are made. It's a difficult choice to make whether we should just inform the developer or warn him, because giving too many warnings might overwhelm the developer, and as such we need to make a call when enough is enough.

Throughout this thesis we have been taking for granted that we've used a certain serialization pattern so that we don't have to deal with nested entities. If we were to **cover the other serialization pattern**, it would present the risk of nested moves and moves to other levels of the tree. It leads to fewer references, but at the same time with nodes of the same kinds being allowed to be put at different levels there are also possibilities of having references to nodes that are not child nodes of the root. Changing the serialization pattern shouldn't be too hard, and with that the importance of this is lowered, but it can still be considered.

There are possible **inconsistencies depending on property values** that can be found. Imagine the situation where you have a class node which has an attribute node, a class with an attribute. The attribute node has a property `value="123"`, and its `type` property points to a `String` datatype. In V1 the `type` is changed to point to an `Integer` datatype, but in V2 the property `value` is given the value `"abc"` instead. In the merged file the class will have an `Integer` attribute with the value `"abc"`, which is not semantically correct. This is probably not a common problem and is probably easy to find too, but it might be interesting to look into.

As mentioned in 4.2.2, we could actually have **multiple references in one property**, where the IDs of the nodes are put after each other in the string, separated by a space character. This was discovered quite late and has therefore not been incorporated enough in the analysis. This will complicate the algorithm further. These kinds of references are not uncommon and should be dealt with.

The fact that these kinds of reference combinations are possible leads us to think that there could be **other anomalies** which we haven't covered in this thesis. This is a very complex area, and as things have been found while working on this thesis it feels like there could be more things to take into consideration. A thorough insight on XMI and different kinds of models is needed to be able to find them all.

As stated above, there are a lot of cases where we don't actually know how common certain changes and combinations of changes are. We can analyze and discuss why certain changes should be more common than other, but there is also the possibility to actually get some **statistics on the most common changes and inconsistencies**. The merge tool could save the amount of different kinds of changes and inconsistencies that occur to give a better view of which parts of the algorithm to prioritize and put effort into. Such information would be very appreciated. Feedback from people working with merging models by using text-based merge tools would also give a good picture of what is most needed.

The algorithm that we have introduced creates a merged output file where we have a lot of annotations and warnings represented by using comments or extensions, as discussed in 3.5.3. There are still problems with the exact **representation of annotations and warnings**, and this could be further analyzed and implemented to show the annotations and warnings graphically in a standalone program or plug-in for an editor.

Annotations and warnings are used for the developer to see changes made and so that he can deal with the merge process easier. While they are not directly part of the model itself, they are still in the file and if they are sent in to the merge tool again they would still be there. Now, they can of course be removed by the merge tool before the new annotations are applied, but creating a **program for removing annotations**[1] after the merge process is done would be a better solution, because then they are not committed to the repository.

Different configurations of this merge tool could be further developed. Giving the developer more opportunities, like choosing one of the input files to be the master version,[2] or to just compare the two versions and get a list of the changes and inconsistencies as output, etc.

The possibility of using **different matching strategies** [2] can also be added. This would mean that the algorithm wouldn't have to satisfy the requirement of having unique IDs for every element in the models.

While we expect the developer to deal with inconsistencies and warnings and such before committing the files to the repository, there is of course cases where this is not done right and we end up with inconsistencies in the repository and then get **input files containing inconsistencies** sent into the merge tool. This scenario can also appear when a developer has updated before committing and ended up in a complicated merge process with inconsistencies and then again updated before all the previous problems are fixed. Since the output from the merge tool is syntactically correct, having a file with inconsistencies as input would work, but it would add more possible problems in the merge process, as we will see below.

The easiest way to deal with this is to simply check the file beforehand if it contains any inconsistencies, and if it does, inform the developer about it and stop the merge. This is what what we have done while the merge tool is still in its experimental phase.

The other way is to allow inconsistencies in the input file and deal with the problems that occur. What would happen then if we get **conflicts with conflicts**? That is, changes that are in conflict with an inconsistency that hasn't been resolved. We can for example have a PU on a property which has already been given two different values, or a move of a node that has been moved to two different parents. There are plenty of cases where the new change can be the same as one of the earlier changes or provide a new value or parent. Should the new change be merged with the earlier conflict or should it be a new conflict with the conflict, and how would that be represented? Which of the inconsistencies can new changes actually be in conflict with? Can one type of inconsistency create another type of inconsistency in combination with a new change? There

[1] One could also include this in the tool or plug-in being implemented.
[2] Always choosing the alternative in the master version in case of conflicts.

could be a need for a lot of work on this area, since it's fairly complicated.

Lastly, there could be made a **more complete implementation** of both the algorithm as well as a test suite. There are still parts that were not implemented and that could pose more problems that we have not found. A complete implementation could also be tested by end users to get feedback on the usability of and need for this kind of merge tool.

Chapter 7

Conclusions

Given the structure and details of the analyzed algorithm presented in Martini [10], this thesis tried to improve it by further analysis and by implementing the algorithm to see if it's possible to implement and to find the problems that appear when doing so.

There are a few restrictions on the tool, which include XMI valid input files, unique IDs for every node in the models and a serialization pattern that only allows a certain type of node at a certain depth from the root node, which makes sure that there are no nested moves. We have also tried to make the algorithm independent from both model type and editors.

We have analyzed the XMI model to find every type of change that can be made on it and every part that can be changed. The change types are additions and deletions of both nodes and properties, moves of nodes, updates of values of properties and reorderings of the child nodes or properties of a node. The last of these were not discussed in Martini's thesis, but have been introduced here. Each of these changes have attributes that can differ from one another. These are the version the change has been made in, the position it has in the model, the change type, the property name and value (if a property is changed), and some other minor things.

Looking at these attributes we have managed to thoroughly go through every combination of these changes to make sure that we find every change in the models and every case of conflicts and syntax violations that these combinations can lead to, as well as many probable context issues. We have also discussed why any conflicts or syntax violation that appears due to a combination of three or more changes always only depends on two of those changes. That is, only combinations of two changes (one in each version) lead to inconsistencies. How to handle these conflicts and syntax violations has also been addressed, and is done by choosing to not apply one or both of the changes in conflict. The problems that can occur due to not applying some changes have also been analyzed and taken care of.

The design in this thesis is presented in more detail than in Martini's, and deals with how exactly to traverse the models and how to apply the changes so that the nodes and properties end up in the right order.

Most of Martini's algorithm was implemented, and worked. We didn't manage to solve the problem of representing change annotations and warnings in the model without breaking the XMI syntax and without interfering with the

rest of the model. This lead to that the algorithm was not finished. The test suite that was to be implemented was not finished either, although the analysis that was done for the algorithm made a very good basis for it.

We hope that the analysis made in this thesis can be used as a foundation for further work in the same area. In the end, there is no easy way to deal with the problem of trying to keep the algorithm model type and editor independent, and there is a need for a standard way of representing the information needed for merged models.

Bibliography

[1] Asklund, U., *Identifying Conflicts During Structural Merge*, Nordic Workshop on Programming Environment Research, Lund, Sweden, June 1-3, 1994.

[2] Barrett, S. C., Chalin, P., Butler, G., *Decoupling Operation-Based Merging from Model Change Recording*, ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems, Oslo, Norway, October 3-8, 2010.

[3] Bendix, L., Emanuelsson, P., *Collaborative Work with Software Models – Industrial Experience and Requirements*, in Proceedings of the Second International Conference on Model Based Systems Engineering – MBSE'09, Haifa, Israel, March 2-6, 2009.

[4] Bendix, L., Koegel, M., Martini, A., *The Case for Batch Merge of Models – Issues and Challenges*, in proceedings of the International Workshop on Models and Evolution – ME 2010, Oslo, Norway, October 3, 2010.

[5] Grose, T., Doney, G., Brodsky, S., *Mastering XMI: Java Programming with XMI, XML, and UML*, Wiley, New York, 2002.

[6] Grune, D., *Concurrent Versions System, A Method for Independent Cooperation*, IR 113, Vrije Universiteit, Amsterdam, 1986.

[7] Jalote, P., *A Concise Introduction to Software Engineering*, Springer, 2008.

[8] Koegel, M., Herrmannsdoerfer, M., von Wesendonk, O., Helming, J., *Operation-based Conflict Detection*, IWMCP '10, July 1, 2010, Malaga, Spain.

[9] Lindholm, T., *A 3-way Merging Algorithm for Synchronizing Ordered Trees — the 3DM merging and differencing tool for XML*, Master's thesis, Dept. of Computer Science, Helsinki University of Technology, 2001.

[10] Martini, A., *Merge of models: an XMI approach*, Master's thesis, LU-CS-EX: 2010-28, Dept. of Computer Science, Lund University, 2010.

[11] Oliviera, H., Murta, L., Werner, C., *Odyssey-VCS: a flexible version control system for UML model elements*, in proceedings of the 12th International Workshop on Software Configuration Management, Lisbon, Portugal, September 5-6, 2005.

[12] OMG, *MOF 2.0/XMI Mapping Specification*, v2.1, 2005.