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

When working in a group of developers, it's preferrable to not lock the files from being edited by more than one developer at a time. Therefore the developers will have to copy the files in the system to their own workspace where they can work on them by themselves. This in itself 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 files, but most of them look at the two versions of a certain file and their common ancestor to see what changes that have been made in the different versions. These changes are applied in the merged version as long as there aren't any conflicts between the changes, like 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. Models are serialized into text form according to XMI, an XML dialect, which describes the syntax of the model. Merge tools that are intended for text-based development can easily break this syntax, making the files unable to be opened in their intended editors. To avoid this, model merge tools often require the developer to take care of the conflicts when merging, in an order decided by the tool, before a merged output file is created. 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 offer a solution where we create a syntactically correct merged output directly, without any input from the user, and include all information about changes and conflicts in the merged file. This type of merge is called a batch merge. The developer can then take care of the conflicts with the flexibility to do so in whichever order he wants, whenever he wants and in whichever editor he wants.

In this thesis we have analyzed, designed, implemented and tested 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 are compared with their common ancestor file to find changes. These changes are compared to find conflicts and inconsistencies. Lastly the changes are applied to a copy of the common ancestor to create a valid XMI file. The algorithm is based on the algorithm in Martini [1], but we have designed a more detailed algorithm than Martini's and tried to implement the algorithm to see if it's possible to implement and to find the problems that appear when doing so.

A model consists of nodes connected as a tree where every node (except the root node) have exactly one parent node, and every node can have any number of child nodes and properties. The properties of a node are identified by their name and they contain the actual data in the model in form of strings. Every node has an ID property for identification.

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 (changing a node's parent node),

updates of values of properties and *reorderings* of the child nodes or properties of a node. Reorderings were not discussed in Martini's thesis, but have been introduced here.

Each change have specific attributes that that describe them. Each change is made in one of the two versions of the model, it has a specific position in the model (it pertains to a certain node), it is of a certain change type, and may pertain to a certain property and in that case have a certain new value for that property, etc. These attributes describe the changes, and their values differentiate them from each other.

Using these attributes as basis we have managed to thoroughly go through and compare every combinaiton of two changes that can be made. We have compared every combination of two changes made in one version of the file to make sure that we can find every change made in the models, and we have compared every combination of two changes made in two different versions to find every inconsistency that can appear. Inconsistencies are either direct conflicts, syntax violations, or probable context issues.

How to handle these conflicts and syntax violations has also been addressed. This is done by choosing to not apply either one or both of the changes in question. This can lead to further inconsistencies, but these cases have also been covered in a satisfactory manner. We have also discussed why any conflict 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, thus we need not look at combinaitons with more than two changes.

In the merged output we wanted to give the developer information about the changes made through annotations, as well as warnings about resolved inconsistencies and probable context issues. Unfortunately, the problem of representing this information in the models without breaking the XMI syntax and without interfering with the rest of the model was not solved. This lead to that the algorithm was not finished.

The design in this thesis is presented in detail. The algorithm is naturally split into three parts. The *Model Comparer* part deals with how exactly to traverse the models and compare the models to find changes, the *Change Comparer* part deals with how to compare the changes to find inconsistencies and resolve these, and the *Change Applier* part deals with how and in which order to apply the changes, annotations and warnings so as to get a satisfacory merged output.

To be able to test the final algorithm, a test suite was implemented, although only partly. The test suite presented is a black box test suite to make it suitable for other model merge tools than the one presented in this thesis. The analysis of the change combinations made a very good basis for this.

There are also a few restrictions on the tool. The input files need to have valid XMI, every node in the models need to have unique IDs and the models need to be serialized in a pattern that only allows a certain type of node at a certain depth from the root node, if you want to make sure that there are no nested moves.

We hope that the analysis and design 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. The type of algorithms used in model merge tools can differ greatly, but the types of changes and change combinations are still the same on the XMI level.

References

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