MDE: Model Differencing

Joeri Reyns

University of Antwerp

Abstract

Model differencing and versioning are important steps during any type of development. Certainly in collaboration based projects it can give a clear view of which structural changes have been made or which implementations have been updated. The problem of determining model differences is intrinsically complex but can be tackled by cohering to the general approach of performing three steps: calculation, representation and visualisation. Both representation and visualisation or often based on either directed or symmetric delta documents, the former uses a procedural way of describing the changes while the latter uses techniques like coloring to show the differences between two models. Both techniques have their merits but are both highly dependent on the information gathered from the calculation step. Simple lexicographical analysis of text files can be implemented in an easy manner but lacks the expressiveness one could want from a differencing algorithm. Model differencing in graphs on the other hand can be reduced to graph isomorphism which is known to be NP-hard. By adding structural knowledge the search space within graphs can be limited but the performance versus expressiveness trade off will always be there. This paper is meant as an overview of the general methodology with some extra information on how model representations could influence the performance, intuitiveness and expressiveness of the difference model.

Keywords: Model Differencing, model based optimisations

1. Introduction

Model differencing and versioning are important steps during any type of development. Certainly in collaboration based projects it can give a clear view of which structural changes have been made or which implementations have been updated. It is known that the problem of determining model differences is anything but trivial but there are generally accepted techniques to limit the search space. Do note that limiting the search space often leads to a trade off between performance and expressiveness. Given the purpose of the application one might prefer performance over expressiveness but in general when developing models one can argue that understanding what impact made differences have is more important. Therefore the search for intuitive and expressive model differencing algorithms still continue.

Email address: joeri.reyns@student.uantwerpen.be (Joeri Reyns)

December 11, 2015

For now this paper provides a general overview of the thought process of how to tackle model differencing problems and how the representation of the models could influence the algorithms. In section 2 the general approach will be discussed based on three steps, calculation, representation and visualisation. Section 3 will provide a deeper insight into the calculation step by giving examples of how different model representations effect the difference calculations. A conclusion will be formulated in section 4 based on what is shown in the previous sections. Lastly we will provide some ideas of future work in section 5

2. General approach

The problem of determining model differences is intrinsically complex [1] but can be tackled by different views on the problem. The most common views to solve model differences are based on constraint satisfaction, search or optimisation views. While constraint satisfaction tries to match certain elements based on their responsibilities it is not always that much different from using search techniques to find corresponding elements. Therefor the representation of the model often gets changed to shift it towards a representation where good and efficient optimisations already exist. This last part will be explained in more detail in section 3.

2.1. Calculation

This step involves finding all the elements and checking which elements changed and which not. This step highly depends on which representation was chosen for the model. Also the actual modelling of the types of changes can be different but generally they all conform to the same thought process. Lets look at [2] to explain the thought process. Here the differences get mapped onto three different kinds of differences:

- Move Differences: Renumbering of elements and differences like rearrangement of attributes or methods of a class.
- Structural Differences: Addition/deletion of attributes or elements, or indication that an element only belongs to one of the original models because there exist no other element with the same structure in the other model.
- Update Differences: This are differences within one single element i.e. attribute value or name changes. This can only occur when the algorithm identifies two attributes as similar. Otherwise it is a structural difference.

In general the calculation step will result in a list of elements contained within the model, and a list of alteration each element has gone through.

2.2. Representation

The Representation of differences can be divided into two main techniques ”directed deltas” and ”symmetric deltas” as explained very well by [3]. Cicchetti et Al. refers to these techniques as visualisation techniques but here they will be presented as ways to represent the alterations of the models. This is done because, although there are two different techniques how to specify changes, there still are many options of how to actually visualise the differences. Both techniques store the differences in a so called delta document. Delta refers to the mathematical interpretation where it means a (small) change compared to the original. In next two subsections we will discuss the differences between both approaches and a deeper description will be provided based on the examples introduced in [3].

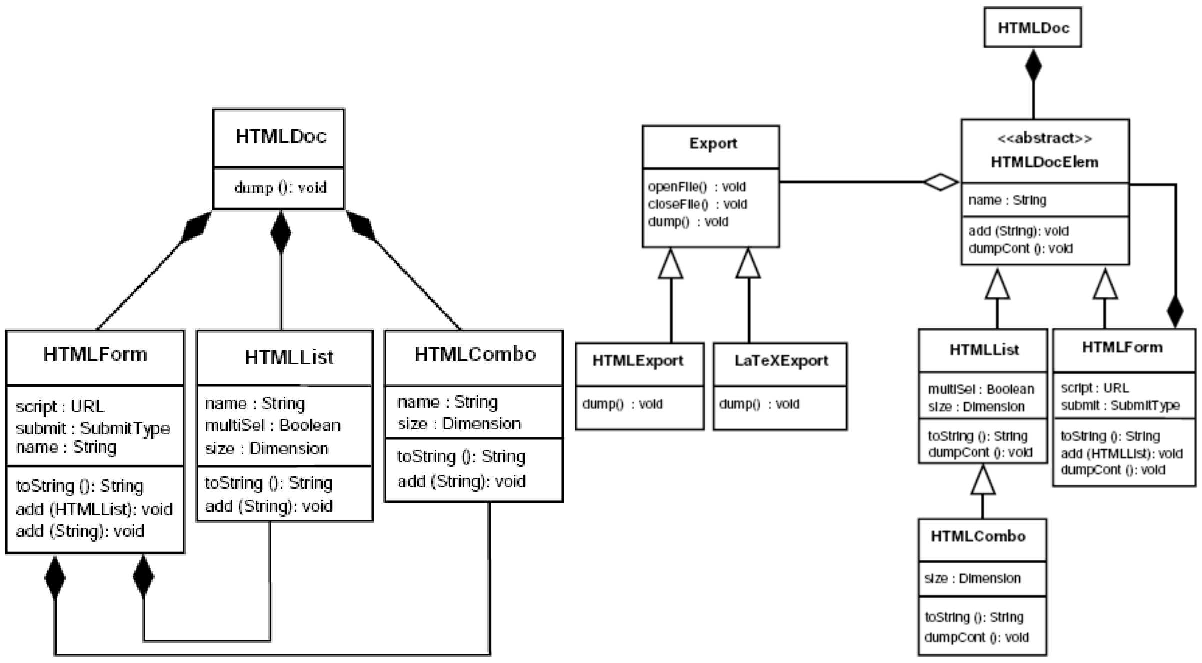

Figure 1: Different versions of a system design

2.2.1. Directed Deltas

Directed delta represents delta documents as the sequence of the operations needed to obtain the new version from the old one. Which is actually a very procedural based way of representing the changes. Like in programming, procedural representation leads to some drawbacks like being quite ineffective to be adopted for documenting changes. But has it merits when trying to model transformation rules. More will became clear after the example:
Edit Scripts represents an implementation of the directed delta approach. Sequences of primitive operations, like add, edit and delete, describe in procedural terms the modifications a model has been subject to. Do note that these operations strongly relate to the calculation algorithm because of what optimisation techniques are used. The biggest advantage of this technique is the compositionality, by applying the composition of deltas we provide the system a way to obtain the new document based on the older version. This property together with the optimisation possibilities makes this technique highly appreciated for its efficiency while readability and intuitiveness are more lacking [3].

2.2.2. Symmetric Deltas

Symmetric delta represents delta documents as the set difference between two compared versions. Also often referred to as coloring techniques which present advantages over procedural methods, for instance differences can be shown as a model which enhances the intuitiveness and can provide the basis for a variety of subsequent analysis. Like directed deltas Cicchetti et Al. also provided an example for coloring techniques:
Coloring techniques permit the modifications to be displayed in a diagram which is the union of the two base models, with the common parts of both base diagrams painted black and the specific elements denoted by color, tags or symbols respectively. It is a symmetric delta approach since it directly compares two versions of a model and highlights the changes. One can see that by showing both models it becomes more beneficial towards the designer because it enhances intuitiveness and readability. However this only works if the base models are not to large and not too many updates are applied to the same elements.

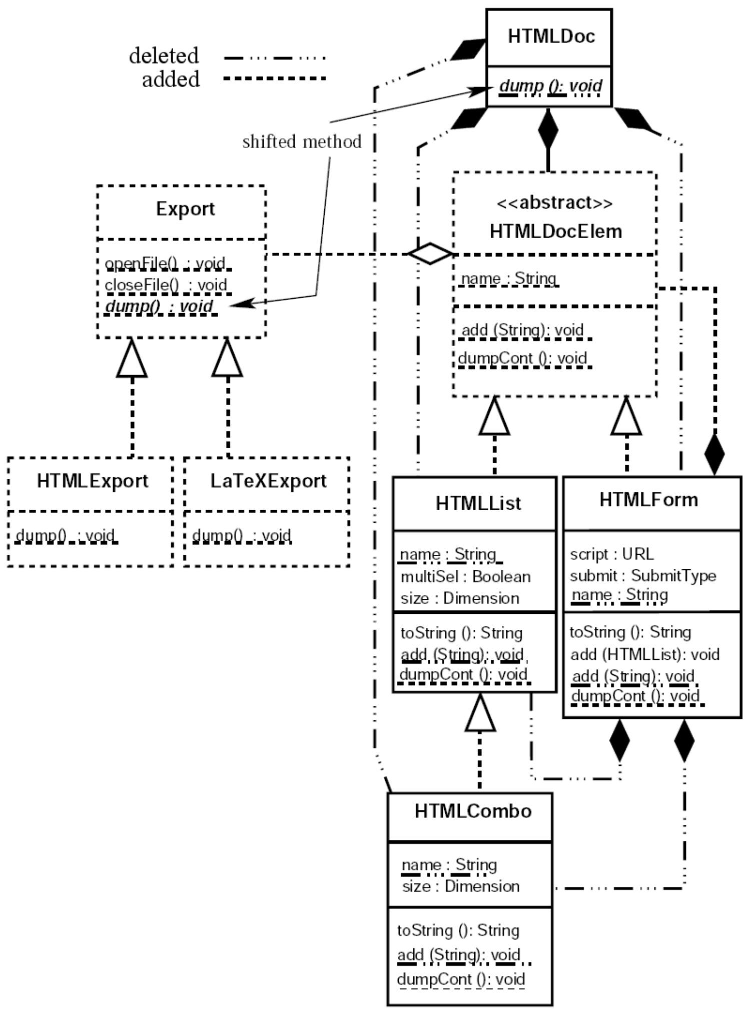

Figure 2: An example of a coloring technique

2.3. Visualisation

Visualisation is an important part of model differencing because it defines how intuitive and readable the differences are based on the expressiveness of the representation technique. The best way would be to develop a concrete syntax which maximises these properties. But concrete syntax development is not really in the scope of model differencing and therefor we will leave it at that.

3. Model representation based optimisations

3.1. String based optimisations

When representing a model as just a document with lines of text the differencing can be done by lexicographical analysis. Which is probably the easiest and most commonly used type of versioning. Both [4] and [5] present existing tools and their downsides. Next follows a citation from [4] which very well captures the string based versioning.
"There has been some work that models the changes of a systems components in terms of CVS-like deltas, which record lines of code that have been added, deleted, or changed, as reported by GNU diff-like tools. These approaches are simple to implement, since it is easy to extract the deltas from a versioning system, such as Concurrent Version System (CVS). Such delta reports are intended to assist software developers in merging different revisions of the system source code. However, GNU diff-like tools are essentially lexical-differencing tools and ignore the high-level logical/ structural changes of the software system. When the intension is to build an accurate evolutionary history of a software system, GNU diff misses a lot of pertinent information. For example, a class renaming or a method movement to another class would most likely be reported as two separate activities: the original entity has been removed and the modified one has been added."
In other words, if we want a more expressive versioning system, we need a way to add domain specific knowledge to the models. Therefor we will further look into some examples based on trees and graphs where the abstraction level enables us to add this kind of information which will result in a performance trade off.

3.2. Tree based optimisations

Given a tree based model representation it becomes apparent that structural differences are easier to find while the complexity does not change that much as shown by [6] where they use an xml based representation (which can be seen as a tree structure) where the easiness of lexicographical analysis can still be used but augmented by extra structural information.

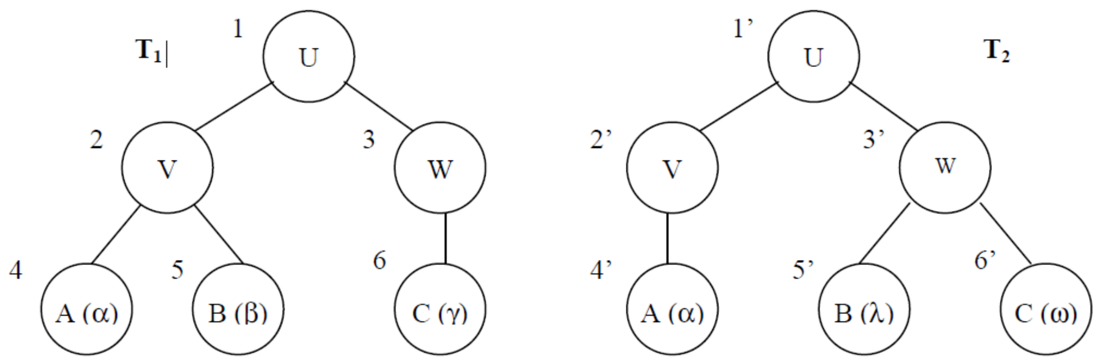

Figure 3: Tree based edit script example

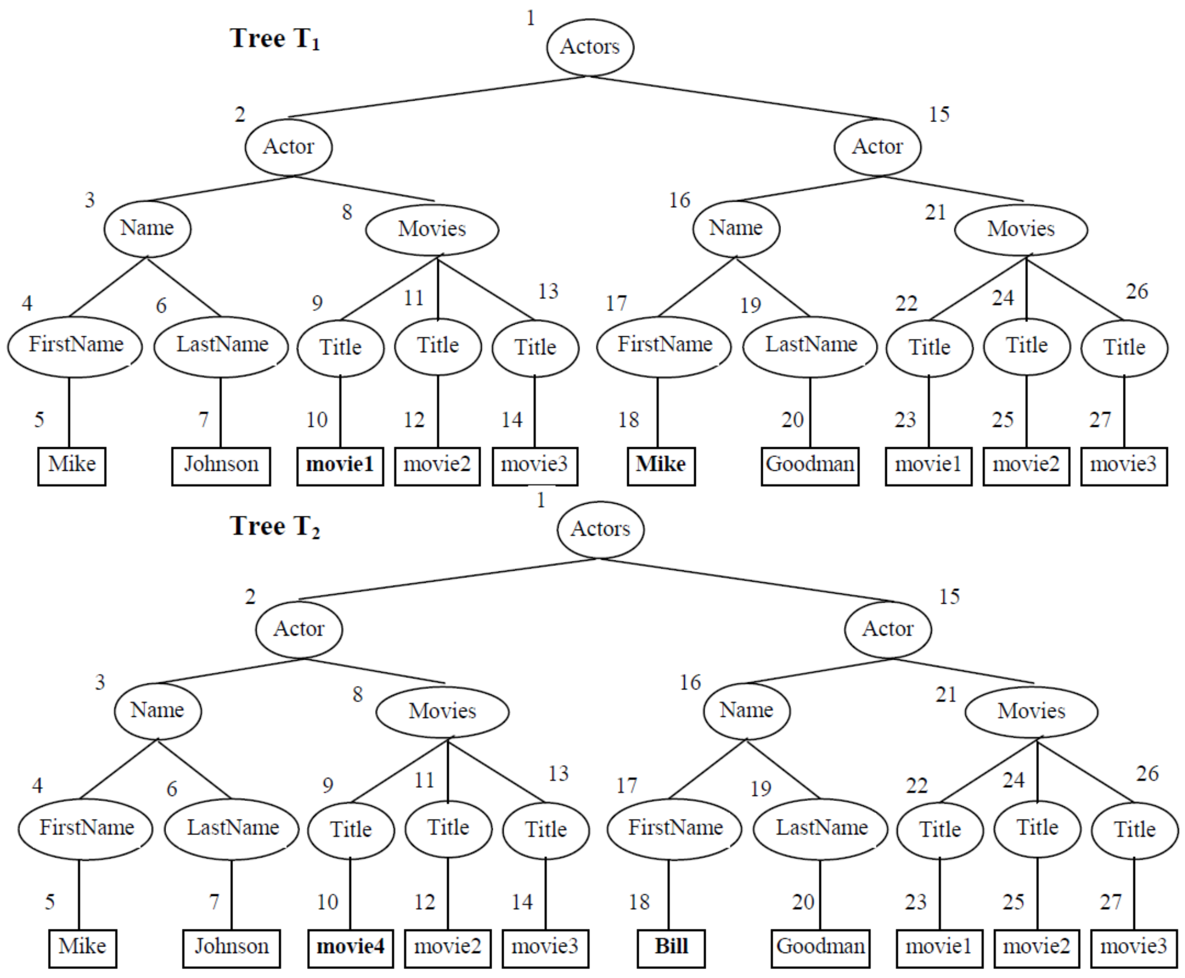

Figure 4: Tree representation example

3.3. Graph based optimisations

Calculating model differences is a difficult task since within graphs it relies on model matching, which can be reduced to graph isomorphism: the problem of finding correspondences between two graphs. Theoretically it is proven that the graph isomorphism problem is NP-hard [1]. But this does not mean that there are no tricks to deal with this complexity. [4] Provides their own graph based solution for java/uml based differencing while [1] lists some known techniques to add more information to the graph in such a way the isomorphism problem gets easier.

- Static Identity-Based Matching: Here it is assumed that each model element has a persistent and non-volatile unique identifier that is assigned upon creation (AToMPM).
- Signature-Based Matching: In this technique, the identity of each element is not static, but instead it is a signature calculated dynamically from the values of its features by means of a (user)defined function.
- Similarity-Based Matching: Above two methods solve the problem in a sort of true / false manner. this category treats models as typed attribute graphs and attempt to identify matching elements based on the aggregated similarity of their features. Note that not every feature is equally important for model matching (e.g. attribute order or class names vs abstract features)
- Custom Language-Specific Matching Algorithms: This category uses the knowledge of particular modelling languages so it can incorporate the semantics of the target language in order to provide more accurate results and drastically reduce the search space.

UMLDiff is an example of a custom language specific matching algorithm and is used for automatically detecting structural changes between the designs of subsequent versions of object-oriented software. It takes as input two class models of a Java software system, reverse engineered from two corresponding code versions. It produces as output a change tree, i.e., a tree of structural changes, that reports the differences between the two design versions [4].

4. Conclusion

There is a general consensus of how to tackle model differencing. Namely first find all elements of the models, then see in what manner they differ from the previous version. Secondly represent these differences in a delta document by either directed or symmetric deltas while lastly visualising the differences in a way that is useful for the problem (e.g. uml diagrams, edit trees, ...). But clearly for the calculation step one still needs problem specific algorithms to find the actual differences. As shown string based differencing has high performance but lacks the expressiveness, while graphs can be just the other way around. Adding extra information information to the graph can help because it limits the search space. Therefor as [3] suggest it would be better to develop a tool which allows differencing based on meta-models and their respective abstract and concrete syntax. Through the ramification process one might be able to add extra information to a graph or tree based representations to correctly identify changes and present them in a domain specific way to take advantage of their expressiveness and intuitiveness. This way a general tool is generated which can tackle the necessary challenges of most domain specific differencing problems. Do note that in essence this still comes down to translating the problem to a different view to solve it in a more general way.

5. Future work

For future work we could further study the UML-Diff tool and apply it to a train simulation model or try to implement a versioning algorithm within AToMPM [7] to cohere with Cicchetti et Al.'s [3] vision of having a versioning tool based on meta-models to provide domain specific information to the differencing algorithms to improve both intuitiveness and expressiveness of the model differences while trying to keep the complexity low enough.

References

[1] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, Richard F. Paige, Different models for model matching: An analysis of approaches to support model differencing.

[2] Jörg Niere, Visualizing differences of uml diagrams with fujaba, FUJABA Days 2004 (2004) 31–34.

[3] Antonio Cicchetti, Davide Di Ruscio, Alfonso Pierantonio, A metamodel independent approach to difference representation, Journal of Object Technology 6 (9) (2007) 165–185. doi:http://www.jot.fm/issues/issues 2007 10/paper9.

[4] Zhenchang Xing and Eleni Stroulia, Umldiff: An algorithm for object-oriented design differencing.

[5] Kyriakos Komvoteas, Xml diff and patch tool.

[6] Yuan Wang, David J. DeWitt, Jin-Yi Cai, X-diff: An effective change detection algorithm for xml documents.

[7] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, H. Ergin, Atompm: A web-based modeling environment., in: Demos/Posters/StudentResearch@ MoDELS, 2013, pp. 21–25.