MultiScience - XXX. microCAD International Multidisciplinary Scientific Conference
University of Miskolc, Hungary, 21-22 April 2016, ISBN 978-963-358-113-1

Merging textual representations of software models

Ferenc Attila Somogyi
MSc, software engineer
Budapest University of Technology and Economics

ABSTRACT

Modeling is an important area of software development. The most prevalent approach to modeling is the graphical approach, which means that models are edited through a graphical interface. In the case of highly complex and large models, however, editing a model through text can be more efficient than the graphical approach. In this paper, a method is presented that can be used to compare and merge models based on their textual representations. This approach helps optimizing teamwork due to its possible application in version control systems for textual representations. Our method is universal, thus it can be applied to any modeling environment and formal language. Differences between the presented method and other, already existing comparing and merging approaches are also described.

INTRODUCTION

Software models (e.g. UML class diagrams [1], or domain-specific models [2]) are often edited through a graphical interface. This approach is convenient, as it is easy to use and understand, although in the case of large and complex models, editing the model through a textual representation can be more efficient, provided that there is an efficient development environment available. The differences between the graphical and textual approach are further elaborated in paper [3]. Transformation between a model and its textual representation can be achieved with the use of a formal language. The textual representation needs to be parsed in order to update the model when the text is changed. It is also necessary to realize the process that generates the textual representation of a model. These tasks are done by the parser and compiler that are associated with the language. It is the parser’s task to ensure the changes done to the text are correctly represented in the model, and the other way around. The parser must also make sure that the model remains consistent during and after the editing process.

Editing a model through its textual representation is similar to traditional source code-based development. In both cases, the editing process is done through texts that are described by a well-defined formal language. The text needs to be parsed in order for the changes to have an effect on the model or to start the compilation process.

In source code-based software development, teamwork is supported by version control systems. These systems help managing and tracking the changes done to source code files. An important function of these systems is being able to compare two files and create a merged output file by solving the conflicts between the two files. This is usually a semi-automated process, which means that user input is necessary to solve all the conflicts, but an effort is made to minimize user input. This functionality can be extended to models and their textual representations.

DOI: 10.26649/musci.2016.060

In this paper, a method is shown that can be used to compare and merge two models based on their textual representations. First, syntax tree-based comparison is introduced and explained. After that, the method is outlined with examples that demonstrate its functionality. It is also shown that this approach is universal, thus it can be applied to any modeling environment and formal language, assuming the parser fulfills certain requirements.

RELATED WORK

During model comparison, the matching of model elements can be done in several ways. Paper [4] sorts these matching approaches into the following categories:

1. Static Identity-Based Matching: it is assumed that each element has a static, unique identifier.
2. Signature-Based Matching: the unique identifier is calculated from the various features of the element.
3. Similarity-Based Matching: calculates the similarity between two elements based on their weighted features.
4. Custom Language-Specific Matching Algorithms: uses the semantics of a specific language to provide the most precise matching that is possible.

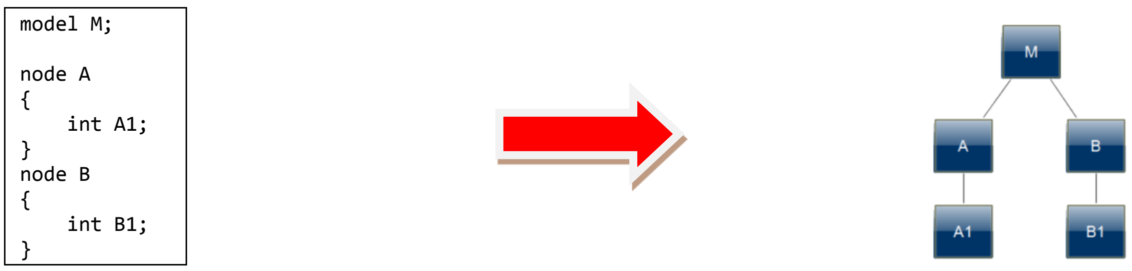
Our method falls into the fourth category, as it uses the parser associated with the language to identify model elements, therefore, the matching is always tailored to the formal language. This is detailed later in this paper.
There are several existing methods that can be used to compare or merge models, but not many that use the textual representations of the model as the basis of the comparison. Papers [5] and [6] describe methods that can be used to compare UML (Unified Modeling Language) models. Both of these methods use similarity-based matching to match the different model elements. These methods compare the internal structure of the models directly instead of their textual representations.
EMF Diff/Merge [7] is a prevalent tool in the industry, used for merging models. It is extensible and configurable, therefore, its matching approach falls into the second category described above. However, the method used by EMF Diff/Merge only applies to EMF-based models, and it does not use textual representations as basis of the comparison.
Paper [8] presents a method that can be used to compare the textual representations of a model. The method is configurable and language-independent, however, it focuses on behavioral models and has difficulties handling moved elements in the text. Therefore, this method cannot be applied to our problem entirely.

COMPARING AND MERGING TEXTUAL REPRESENTATIONS

In this section we present a method that is used to compare and merge the textual representations of models. The method is presented with simple examples to better demonstrate its functionality.

Introduction to syntax tree-based comparison

During the parsing process, the parser builds an abstract syntax tree (AST) from the text. The AST can then be processed instead of the raw text. Book [9] contains more information regarding the parsing process, including information on abstract syntax trees. In this paper, examples that are shown use a theoretical language that describes our theoretical models to make the examples easier to understand. This theoretical language along with the process of building an AST is illustrated in Figure 1.

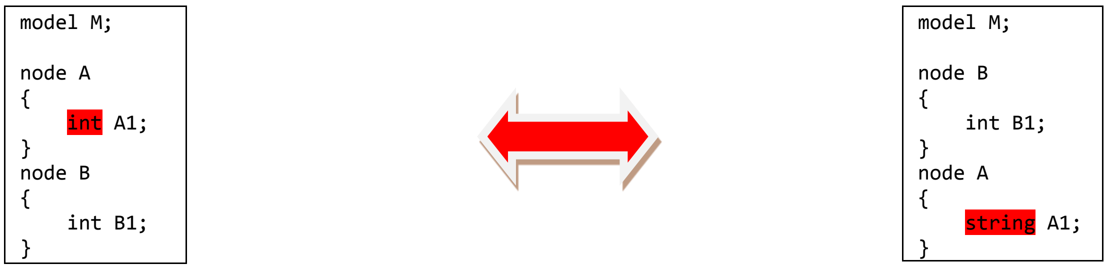


```
model M;

node A
{
    int A1;
}
node B
{
    int B1;
}
```

Figure 1.
Building an AST

For the sake of simplicity, let us assume that our theoretical models can only contain nodes and these nodes can have any number of typed attributes. Working with edges and other model elements is very similar to nodes, but in our examples we are only going to use nodes and attributes as model elements. In our theoretical models, every model element is identified by its unique name. Therefore, two elements cannot have the same name.
The textual representation of a theoretical model is also depicted in Figure 1. Our sample model is named M. The model has two nodes, A and B. A has an integer attribute A1 and B has an integer attribute B1. Figure 1 also describes the AST built from this textual representation. We are going to use the same notation system in our future examples.
Abstract syntax trees are also useful when trying to compare two textual representations of a model. Comparing the textual representations results in less information as opposed to comparing the abstract syntax trees built from the textual representations. This is demonstrated in the next example.



```
model M;

node A
{
    int A1;
}
node B
{
    int B1;
}
```

```
model M;

node B
{
    int B1;
}
node A
{
    string A1;
}
```

Figure 2.
Comparing the textual representations

There are two differences between the two textual representations illustrated in Figure 2. Firstly, the order of the two nodes (A and B) are reversed. Secondly, the type of attribute A1 is different. If we use a traditional text comparing method (like [10]), we get a result that shows the differences between the two texts. The result shows that the texts associated with nodes A and B are present in one text, and missing in the other, and the other way around due to the symmetry of the algorithm. However, the results do not show that the texts represent nodes A and B, thus this conflict between the two texts cannot be handled or displayed to the user correctly. The situation is the same with the attribute type of A1.

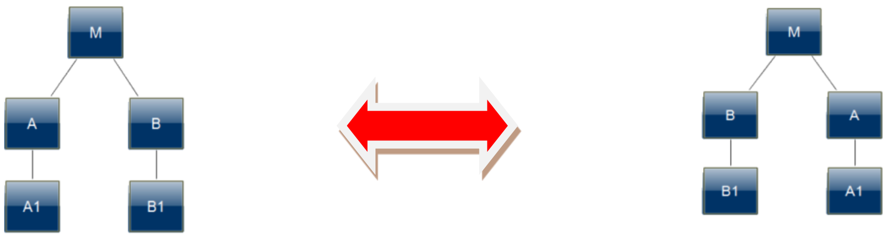


Figure 3.
Comparing the abstract syntax trees

Figure 3 shows the comparison of the abstract syntax trees that are built from the texts depicted in Figure 2. The order difference between nodes A and B is clearly visible from the structure of the syntax trees. The type difference of attribute A1 is discovered once the texts that represent A1 are compared. Therefore, only comparing the syntax trees is not enough to discover every conflict between two textual representations, regular textual comparison is still needed. However, using the syntax trees makes the conflicts between the two texts manageable and easily displayable to the user.

Differences between source code comparison and the appointed problem

Using syntax trees during the comparison of source code files is also possible, however, there is a substantial difference between comparing source code files and comparing the textual representations of a model. In the case of a single source code file, syntactic correctness can be demanded, but we cannot be certain that the code is always semantically correct. While working with models however, we can demand both syntactic and semantic correctness from the textual representations. This is supported by the fact that most modeling environments (e.g. [11], [12] and [13]) guarantee that models edited through the graphical interface remain semantically correct. The same policy can be extended to the textual representations of models, since we are working with the models in this case as well. Since we demand that the texts are both syntactically and semantically correct, the abstract syntax trees built from the texts are always correct as well. Therefore, the abstract syntax trees can always be used for the comparison and it always gives correct results.

Universality of the method

The aim for this method is to be as universal as possible, which means that it should be applicable to any modeling environment and formal language. We are using a syntax tree-based comparison, thus the method can be applied to any modeling environment, since it is not dependent on the inner structure of the models. In order for the method to be applicable to any language, the parser associated with the language has to delegate a number of operations that are used during the comparing and merging process. These operations are the following:

1. Parsing the text and building the AST.
2. Syntactic and semantic verification of the text.
3. Deciding if two trees represent the same element.
4. Deciding if the difference between the texts of two trees concerns only the format.

The first two operations are needed to use syntax tree-based comparison and to ensure the syntactic and semantic correctness of the texts and therefore the models they represent. The third and fourth operations are used during the comparing process. They are used to decide which trees represent the same elements and also to decide if the difference between two texts representing the same tree concerns only the format of the text. This typically means difference in white space characters or comments, but it is ultimately up to the parser to decide what differences are of concern to the format only. These type of differences can be treated differently during the comparing process. All of the operations mentioned above (with the possible exception of the fourth one) are typically already realized in the parser, therefore delegating them is usually not a very costly procedure.

Outline of the method

The first step of the comparison is to establish which trees represent the same elements. This is done using the parser's delegated operation that decides if two trees represent the same element. The algorithm tries to match every tree on the same level (due to performance optimization) of a syntax tree with each other. Every tree that has no matches is stored and is matched with each other at the end. Trees that have no matched counterpart are marked as new trees. An example that presents this algorithm is illustrated in Figure 4.

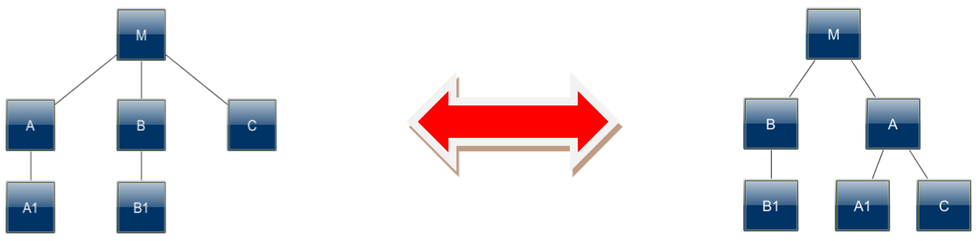


Figure 4.
Tree matching

In our theoretical models, two elements are considered the same if their name match. Therefore, the parser decides that two trees represent the same element if their names match. On the first level, A and B are matched while C in the left tree cannot be matched. On the third level of the tree, A1 and B1 are matched while C in the right tree cannot be

matched. Finally, at the end, the algorithm attempts to match the two C trees and it succeeds. Therefore, every tree is marked as matched in this example.
The second step of the comparison is to discover every conflict that is present between the two syntax trees. Conflicts are sorted into the following categories:

1. *Different text*: two trees represent the same element, but their texts are different. The parser decides whether the difference concerns only the format of the text. In order to recognize this type of conflict, the texts of every matching tree pair has to be compared with each other.
2. *New tree*: there is a tree that can be found in one of the syntax trees, but not in the other. These are trees that have no matched trees associated with them after the tree matching, therefore this conflict type can be easily recognized.
3. *Order*: the order of trees that are found in both syntax trees are in a different order. Recognizing and handling this type of conflict can be handled in a variety of ways depending on the implementation of the method.

The last step of the algorithm is the merging process. First, every conflict has to be solved. A solution is a text that is assigned to a conflict and will replace the conflict during the merging process. Solving the conflicts are done in two phases. The first phase is the automatic phase, during which the method assigns automatic solutions to every conflict when possible. The second phase is the manual phase, during which the user can choose a solution for every conflict. The automatic solutions chosen by the algorithm can be overwritten in this phase. We have to take into consideration that the different conflict types interact with each other with regard to their place in the merged text. This can be solved by examining these interactions and using absolute position in the merged text for every conflict.
The merging process can be initiated by the user once every conflict has a chosen solution. During the process, the conflicts are replaced with their chosen solution and the merged text is created. Figure 5 and Figure 6 describe the merging process in a final example.

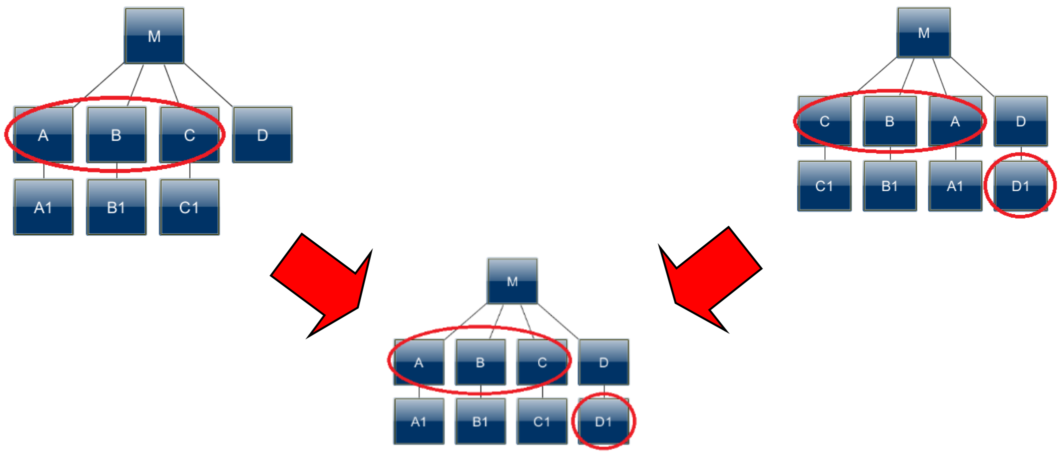
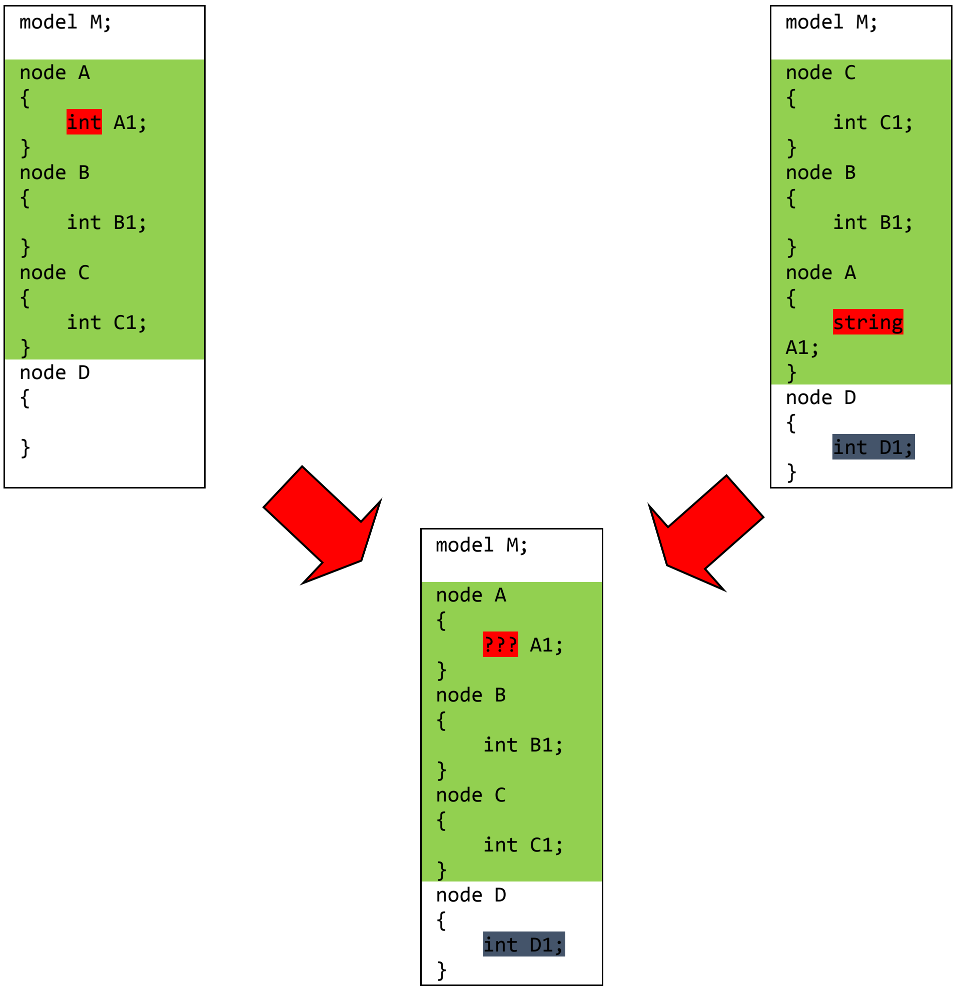


Figure 5.
Merging example – syntax trees



```
model M;

node A
{
    int A1;
}
node B
{
    int B1;
}
node C
{
    int C1;
}
node D
{

}
```

```
model M;

node C
{
    int C1;
}
node B
{
    int B1;
}
node A
{
    string
A1;
}
node D
{
    int D1;
}
```

```
model M;

node A
{
    ??? A1;
}
node B
{
    int B1;
}
node C
{
    int C1;
}
node D
{
    int D1;
}
```

Figure 6.
Merging example – texts

There are three conflicts between the two syntax trees and their related texts. Firstly, the order of nodes ABC in the left tree is CBA in the right one. Secondly, attribute D1 is only present in the right syntax tree. And lastly, the type of attribute A1 is different in the two texts. In the example, we assume an implementation of the method that automatically chooses the left order as a solution for every order conflict and automatically inserts the new tree for every new tree conflict. The type difference of A1 is semantic, thus we cannot decide on an automatic solution. The automatic solutions entirely depend on the implementation of the method, and can be changed or refined.
During the manual phase of solving the conflicts, the user can change the solutions for every conflict. In this case, changing the solution of the order conflict from ABC to CBA has a side effect of changing the position of A1, thus changing the position of the conflict associated with A1 as well. These interactions have to be examined and handled for the method to work as intended.

CONCLUSION

There are very few existing methods that use the textual representations of models as basis of the comparison.

The method outlined in this paper can be used to compare and merge the textual representations of models. Syntax tree-based comparison is used in order for the comparison to be precise. Using syntax trees also enables it to work with every modeling environment as the method is not dependent on the inner structure of the actual models. With a few restrictions applied to the parser, this approach can be used with any formal language. Thus, the outlined method is a universal and precise tool that can be used to compare and merge the textual representations of models.

REFERENCES

[1] UML class diagrams: http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

[2] M. Fowler: **Domain Specific Languages**. Addison-Wesley Professional, 2010

[3] H. Grönniger, H. Krahn, R. Bernhard, S. Martin and V. Steven: **Text-based Modeling.** Proceedings of the 4th International Workshop on Software Language Engineering (ateM 2007), Nashville, TN, USA, October 2007 Informatik-Bericht Nr. 4/2007, Johannes-Gutenberg-Universität Mainz, October 2007

[4] D. S. Kolovos, A. Pierantonio, D. D. Ruscio and R. F. Paige: **Different models for model matching: An analysis of approaches to support model differencing.** In Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (CVSM '09). IEEE Computer Society, Washington, DC, USA, 1-6.

[5] Z. Xing and E. Stroulia: **UMLDiff: an algorithm for object-oriented design differencing.** In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05). ACM, New York, NY, USA, 54-65., 2005

[6] U. Kelter, J. Wehren and J. Niere: **A Generic Difference Algorithm for UML Models.** In P. Liggesmeyer, K. Pohl & M. Goedicke (eds.), Software Engineering (p./pp. 105-116), : GI. ISBN: 3-88579-393-8, 2005

[7] EMF Diff/Merge: http://www.eclipse.org/diffmerge/

[8] R. v. Rozen and T. v. d. Storm: **Model Differencing for Textual DSLs.** BENEVOL 2014-Proceedings of the Belgian-Netherlands Evoluation Workshop.

[9] A. V. Aho, M. S. Lam, R. Sethi and J. Ullman: **Compilers: Principles, Techniques, and Tools**. Addison Wesley, 2006

[10] GNU Diff Utils: https://www.gnu.org/software/diffutils/

[11] Eclipse Modeling Framework: https://eclipse.org/modeling/emf/

[12] Generic Modeling Environment: http://www.isis.vanderbilt.edu/projects/gme/

[13] Visual Modeling and Transformation System: https://www.aut.bme.hu/Pages/Research/VMTS/Introduction