Formal Description and Verification of a Text-based Model Differencing and Merging Method

Ferenc A. Somogyi and Mark Asztalos
Department of Automation and Applied Informatics,
Budapest University of Technology and Economics,
1111 Budapest, Hungary

Keywords: Version Control, Model Differencing and Merging, Text-Based Modeling, Algorithm, Verification.

Abstract: Version control is an integral part of teamwork in software development. Differencing and merging key artifacts (i.e. source code) is a key feature in version control systems. The concept of version control can also be applied to model-driven methodologies. The models are usually differenced and merged in their graph-based form. However, if supported, we can also use the textual representation of the models during this process. Text-based model differencing and merging methods have some useful use cases, like supporting the persistence of the model, or having a fallback plan should the differencing algorithm fail. Using the textual notation to display and edit models is relatively rare, as the visual (graph-based) representation of the model is more common. However, many believe that using them both would be the ideal solution. In this paper, we present the formal description of a text-based model differencing and merging method from previous work. We also verify our algorithm based on this formal description. The focus of the verification is the soundness and completeness of the method. The long term goal of our research is to develop a modeling environment-independent algorithm. This could be used in version control systems that support textual representations.

1 INTRODUCTION

Version control is an integral part of teamwork in traditional software development. Version control systems (Spinellis, 2005) are crucial in keeping teamwork organized. The concept of version control can also be applied to model-based development techniques in order to achieve greater efficiency during teamwork. This is a young research area. Using version control greatly benefits model evolution and management (Paige et al., 2016), as we can keep better track of evolving (changing) models during development.

Difference and merging different versions of the same code is a key feature in version control systems. Model Differencing and Merging (MDM) is different from source code differencing and merging. In the former case, the artifacts are graph-based models, while source code is usually text-based. This means that different approaches are needed for the two cases. It is worth noting that our focus is on Domain-Specific Modeling (DSM) (Kelly and Tolvanen, 2008), where the models are almost always in a graph-based form, though there might exist other areas where they are not. In DSM, models are usually processed by model transformations (Sendall and Kozaczynski, 2003), with the next step usually being code generation. These practices already have a well-established literature and practical applications (Bergmann et al., 2015). Model differencing and merging is a research field with academical results (Lin et al., 2004) (Altmanninger et al., 2009) and some industrial applications (Brun and Pierantonio, 2008), but it is mostly considered a young research field. Text-based MDM methods are even rarer in existing research.

Table 1: Graphical and textual approaches in DSM.

Graphical approach	Textual approach
Broad view	Detailed view
Easier to read	Easier to write
Simulation / animation	Handling larger models
Domain experts prefer it	Developers prefer it
-	Serialization support

The most common approach used to display and edit models is the graphical (visual) approach. There are some approaches that connect graphical and textual languages (Eysholdt and Behrens, 2010), but these are in the minority. Displaying and editing mod-

Somogyi, F. and Asztalos, M.
Formal Description and Verification of a Text-based Model Differencing and Merging Method.
DOI: 10.5220/0006728006570667
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 657-667
ISBN: 978-989-758-283-7

els in a textual form can have many benefits, both inside and outside (Petre, 1995) (Moher et al., 1993) modeling-related applications. Table 1 summarizes the main advantages of these approaches. Many argue that using the two together is the ideal solution, because we can keep the advantages of both (Grönniger et al., 2007) (Pérez Andrés et al., 2008). This raises the problem of keeping the two notations synchronized (van Rest et al., 2013). Text-based MDM methods can help solve this problem as we can use them while reloading a previously-edited textual representation to identify changes since the last edit.

In previous work (Somogyi, 2016) (Somogyi and Asztalos, 2016), we presented our own text-based MDM method. The algorithm was developed for a specific modeling environment (VMTS) and a specific language (VMDL) used to describe the textual representations. Our goal was to develop a method that can be used to merge arbitrary VMTS models based on their VMDL representations. VMTS is a modeling tool used for educational and industrial purposes. As an industrial example, VMTS was used in the graphical programming of programmable logic controllers (PLC) under the IEC standard. Our method can be applied to both industrial and educational purposes.

The main application of our approach is version control. We can also use it to aid synchronization by recognizing the changes that occurred between two editing sessions of a textual representation. This is useful to keep the textual representation synchronized with the stored model when the model is changed by other means (graphical edit, direct edit, etc.). It can also be used alongside real-time synchronization as an extra layer. Our motivations for using and researching text-based MDM methods is further detailed in Section 2.2.

The goals of this paper are to formally present our approach and to provide its verification. During the verification, we examine the soundness and completeness of the algorithm. The meaning of these concepts vary based on the phases of our approach. The paper is structured as follows. In Section 2, we briefly introduce the modeling environment our method was developed for, and present the language that describes the textual representations. We also talk about our motivations behind researching text-based MDM methods. Section 3 is the main contribution of the paper. It contains the formal presentation and the verification of our approach. The analysis is divided by the three phases of the algorithm. Finally, Section 4 concludes the paper and contains plans for future work.

2 BACKGROUND

In this section, we introduce the modeling environment used by our algorithm. We also present the language responsible for describing the textual representation of the models. Next, we briefly (and informally) introduce our text-based model differencing and merging (MDM) method that we published in previous work. We also talk about our motivations behind researching text-based MDM methods.

2.1 VMTS and VMDL

The Visual Modeling and Transformation System (VMTS, 2003) (Levendovszky et al., 2005) is a graph-based, domain-specific (meta)modeling and model processing framework. The system provides a graphical interface for defining, customizing, and utilizing languages. VMTS supports N-level meta instantiation instead of the often-used meta levels of the MOF specification (Meta Object Facility, 2003). The entities (nodes) and relationships (edges) in VMTS are identified by a globally unique identifier (GUID). Both the nodes and the edges can have typed at-


```
[# 23b087c0-7ace-4eaf-ac75-dbe09aa94439]
[RootMeta]
model LibraryMeta;
InstanceName : "BookModel";

[# 14d650ad-55e5-4a2c-b135-7736b1430e85]
[Entity]
node BookMeta
    attribute Title "String" [1..1];
end

[# 1daf358e-35d3-4573-b91a-c2bd52b28e31]
[Entity]
node AuthorMeta
    attribute AuthorName "String" [1..1];
end

[# 333678e5-2036-4ba7-90f5-4cee250e082e]
[Relationship]
edge BookAuthorRelationShipInstance
    LeftMultiplicity : [1..1];
    RightMultiplicity : [1..1];
    InstantiationBehavior : "MetaRelationship";
    LeftConnector : BookMeta;
    RightConnector : AuthorMeta;
    LeftRoleName : Books;
    RightRoleName : Authors;
end
```

Figure 1: A VMTS model and its VMDL representation.

tributes. The nodes can contain other nodes. The edges belong to one of the following categories: association, composition or inheritance. The framework supports many common modeling concepts, like multiplicity, cardinality, etc.

The Visual Model Definition Language (VMDL) is a language that describes VMTS models in a textual form. The models can be edited in this textual form. We achieve this by using a formal grammar to describe the language. VMDL uses a grammar implemented in ANTLR (Parr, 2013). When the model is updated via the text, an abstract syntax tree (AST) is parsed (Aho et al., 2005) before updating the model. The AST contains more semantic information than the raw text and we can update the model based on this information. We can also keep the non-semantic information (comments, white spaces, etc.) in the text separate from the semantic information. Figure 1 illustrates a sample VMTS model and its textual representation in VMDL. The example is a simple metamodel for a library that contains books and authors. The books have a title, and the authors have a name attribute. There is a many-to-many relationship between the books and authors.

2.2 Our Text-based MDM Approach

Our text-based MDM method is capable of differencing and merging two different versions of a VMTS model described by the VMDL language. The algorithm performs the differencing and merging based on the trees parsed from the raw text. It also uses the raw texts during the process. The algorithm consists of three main phases:

- **AST matching.** The algorithm matches every subtree pair in the trees parsed from the two versions of the model.
- **Conflict detection.** The algorithm detects every difference (conflict) between the matched subtrees. Unmatched trees are also a source of conflict.
- **Merging.** The algorithm merges the two versions into a merged model based on the trees, the raw texts, and the discovered conflicts. The end result must always be a syntactically and semantically correct model.

Figure 2 and Figure 3 illustrate the difference between a traditional text differencing and merging algorithm, and our own approach. The figures contain two versions of the previously described book meta model in VMTS. The models are represented in their textual form. There are two differences between the two versions: 1) the multiplicity of the *Title* attribute in the *BookMeta* node is different, and 2) the order of the model elements is different. Figure 2 shows the result of the differencing process in the case of a traditional text differencing and merging approach (KDiff3, 2003). The algorithm could not recognize the movement of the *BookAuthorRelationshipInstance* edge, since the algorithm only works on the level of the raw text. Instead, it recognized the other two model elements (*BookMeta* and *AuthorMeta*) as differences. Moreover, since the position of *BookMeta* changed in the text, it could not recognize the difference in the *Title* attribute either. To sum it up, the algorithm could not handle the differences on the level of the model, the result was not accurate. On the other hand, our method can recognize these differences correctly. The result is illustrated in Figure 3. The identifiers of the elements are omitted from the figure, as they are not relevant now. Both the movement of the edge, and the multiplicity of the *Title* attribute is easily recognizable based on the trees and the raw text. They can also be tracked to the respective model elements in the model.

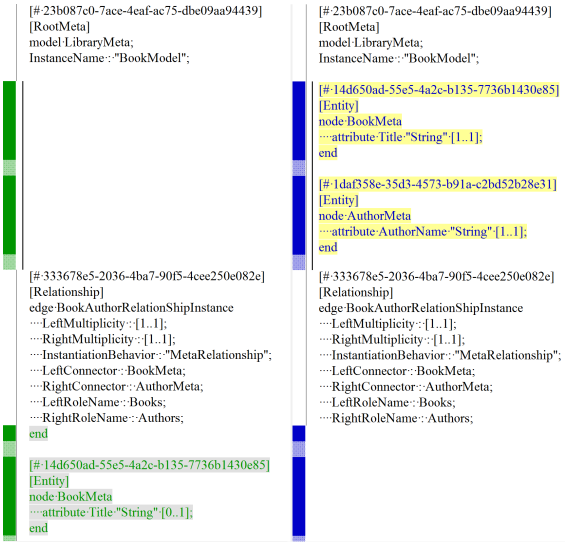

Figure 2: Traditional text-based differencing.

2.2.1 Motivations Behind Text-based MDM

The main use cases for text-based MDM methods (including our approach) are as follows:

- When using a traditional text differencing tool (i.e. KDiff), we cannot recognize the conflicts on the semantic level of the model. We can recognize the changes in the text, but not on the level of the model elements. A common example of this problem is the recognition of moved elements in the text. We have seen an example of this in Figure 2.

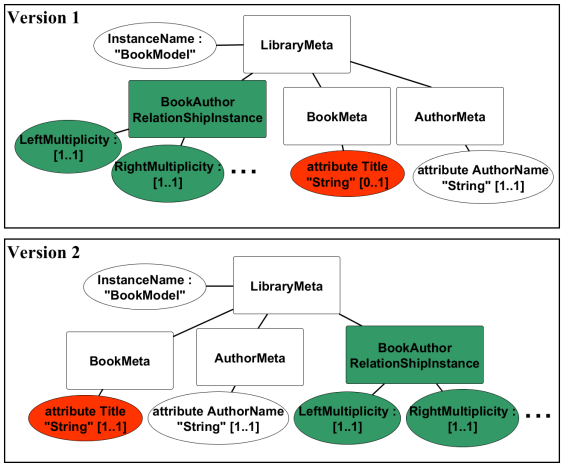

Figure 3: Text-based AST differencing.

- By describing our models in a textual form, we can use this form to support serialization instead of a standard XML-like format like XMI (XML Metadata Interchange, 2015). The main advantage of this is the easier readability of the text, especially during version control. Using a text-based MDM method further supports this process.
- It is beneficial to preserve the non-semantic information (i.e. comments, white space) in the text after reloading the model. Text-based MDM methods support this.
- Text-based MDM methods can help synchronization by recognizing changes that occurred by other means (direct edit, graphical edit, etc.) between two editing sessions of a textual representation.
- If a text-based MDM method cannot find every difference between the two versions based on the trees, we can always rely on simple text-based differencing. The differences might not be accurately recognized, but the end user is always informed of there being a difference. This is an important advantage, as this makes text-based approaches less error-prone compared to structure-based approaches. Structure-based approaches usually do not have a fail-safe like this.

We also have our own personal motivations behind researching text-based MDM methods:

- There are few existing text-based MDM methods (van Rozen and van der Storm, 2015). We are interested in seeing how they compare to structure-based MDM methods according to different aspects.
- Our long-term goal is to develop a text-based MDM method that is independent from VMTS and VMDL, meaning that it can be used with other modeling environments and languages. Parts of the algorithm are already developed with this goal in mind.

2.2.2 Comparison with Existing Approaches

Our method uses the abstract syntax trees parsed from the textual representation during the differencing and merging process. This approach is similar to existing AST-based differencing tools, like GumTree (Falleri et al., 2014) or ChangeDistiller (Gall et al., 2009). However, our method is based on a different concept than most AST differencing approaches. Most of the time, these tools focus on source code differencing. They usually focus on a specific programming language, like Java in the case of GumTree and ChangeDistiller. In contrast, our approach is tailored for modeling, more specifically, the textual representations of graph-based models. Among other smaller differences, this is most apparent in two main parts of the algorithm:

- The AST matching phase uses a matching operation that is based on the parser of the language used to describe the textual representations. This makes the approach more customizable, as it can be tailored to multiple modeling languages in the future.
- At the end of the merging process, our approach checks if the merged model is both syntactically and semantically correct. This is also done with the help of the parser, as we have to build the model from the text to perform this check.

While it might be possible to apply these AST differencing tools to our problem, it is usually not feasible. For example, our accuracy (ratio of correctly identified conflicts) would suffer, since we are not using an approach tailored to modeling. Checking the correctness of the model would also be difficult without using the parser to build the model from the text. Modeling-focused approaches like our method are better suited for model differencing and merging problems. In theory, our text-based MDM method can also support other modeling languages VMTS and VMDL. In addition, our approach supports the correctness check of the model at the end of the merge phase. This is an important constraint, as most modeling environments do not support saving incorrect models (Steinberg et al., 2008) (Levendovszky et al., 2005) during the editing process. Thus, in our opinion, applying this constraint to model-based version control is recommended.

3 VERIFICATION OF THE ALGORITHM

This section is the main contribution of the paper. It contains the formal description and verification of our text-based MDM method. The concept of our approach was presented in previous work (Somogyi, 2016) (Somogyi and Asztalos, 2016). In this section, we formally present our method using pseudo code, and verify it according to different aspects. The verification is divided by the three main phases of the algorithm. We examine the soundness and completeness of the different phases of the algorithm. In the proofs, we use the notations of the pseudo code.

3.1 AST Matching Phase

The input of this phase are the two abstract syntax trees (AST) parsed from the textual representations. The output is a list of matched pairs and unmatched subtrees. A subtree is a node in an AST. A subtree itself is considered an AST as it can also have children. During the AST matching phase, the algorithm matches every subtree in the two abstract syntax trees. It first tries to pair every subtree on the same level with each other. We use this heuristic, because in practice, most matches will be found on the same level (nodes, attributes, etc.) of the trees. Moreover, subtrees being moved on the same level (i.e. change of order in the text) are common when we edit the textual representations of models.

Input: AST_1, AST_2
Output: MP, $U_1 \cup U_2$
if $\neg$ *(IS_MATCH(AST_1,AST_2))* **then**
  return $\emptyset$;
end
$C_1 \leftarrow CHILDREN(AST_1)$;
$C_2 \leftarrow CHILDREN(AST_2)$;
$Tried \leftarrow \emptyset$;
$(MP, Tried) \leftarrow$
 $MATCH\_ASTS(C_1, C_2, Tried)$;
$U_1 \leftarrow UNMATCHED(MP, AST_1)$;
$U_2 \leftarrow UNMATCHED(MP, AST_2)$;
for $\forall i \in U_1$ **do**
  for $\forall j \in U_2$ **do**
    if $(i,\ j) \notin Tried$ **then**
      (MP, Tried) ← (MATCH_ASTS(i, j, Tried));
    end
  end
end

Algorithm 1: AST matching - main algorithm.

Input: C_1, C_2, $Tried$
Output: $Pairs$, $Tried$
for $\forall i \in C_1$ **do**
  for $\forall j \in C_2$ **do**
    if $(i,\ j) \notin Tried$ **then**
      $Tried.ADD((i,\ j))$;
      if *IS_MATCH(i, j)* **then**
        $Pairs.ADD((i,\ j))$;
        $c_i \leftarrow CHILDREN(i)$;
        $c_j \leftarrow CHILDREN(j)$;
        Pairs.ADD(MATCH_ASTS (c_i, c_j, Tried));
      end
    end
  end
end

Algorithm 2: The MATCH_ASTS operation.

Algorithm 1 illustrates the main algorithm of the AST matching phase. First, we check if the roots of the two trees can be matched using the IS_MATCH operation. This is a configurable user function that determines if two subtrees (in this case, the roots) represent the same element. This functionality is provided by the parser of the language (i.e. VMDL). The operation currently supports VMDL and VMTS, but in theory, it can be extended to support other modeling languages. After checking the roots, we call the $MATCH_ASTS$ subroutine that is responsible for matching the children of two subtrees, followed by their children, etc. It always tries to match elements on the same level. The $CHILDREN$ operation returns the children of a subtree. In this case, we use it to get the children of the root trees. We also store subtree pairs that we previously tried in the matching process ($Tried$) , in order to avoid checking them twice. After the subroutine returns, the algorithm gets every unmatched tree, and tries to match them with each other, using the subroutine again. This step is necessary to find matches that are not on the same level of the trees. The $UNMATCHED$ operation returns every unmatched tree in an AST. In practice, a common way to do this is by labeling the unmatched trees.

Algorithm 2 depicts the subroutine used by the main algorithm. The input of the subroutine are two subtree lists (C_1, C_2) that represent trees on the same level, and the list of already tried pairs. The output are the matched pairs and the unmatched subtrees. The subroutine tries to match a pair, following it up by doing the same with their children. It always checks subtrees on the same level to see if they match. The matching is done using the aforementioned IS_MATCH operation. We also avoid matching a pair that we have tried before ($Tried$).

The basic algorithm for this phase would compare every subtree with every other subtree in the other AST. The difference between the basic algorithm and our approach is the same-level heuristic mentioned above. This greatly benefits the performance of the algorithm when it comes to VMDL. We believe that this is also true for most formal languages. As we have mentioned before, the reason for this is that in practice, nodes, attributes, etc. tend to be on the same level in the abstract syntax trees. Thus, in practice, using the heuristic usually results in better performance.

3.1.1 Soundness

We consider that the AST matching phase of a text-based MDM algorithm is sound, if it does not contain any incorrectly matched pairs on its output. In this subsection, we prove that the AST matching phase of our algorithm is sound, assuming some limitations.

Theorem 1. *The AST matching phase of our MDM approach is sound, assuming the matching function IS_MATCH always returns the correct result.*

Proof. The algorithm collects the matched pairs in variable MP. Pairs to MP are always added within the $MATCH\_ASTS$ subroutine: $Pairs.ADD((i,\ j))$. This addition is always preceded by the IS_MATCH operation. Since we assumed that the IS_MATCH operation always returns the correct value, the algorithm can never match incorrect pairs, thus, it is sound. □

Remark. *We assume that the IS_MATCH operation always returns the correct result, because it is a configurable part of our algorithm. In theory, it means that it can be extended to work with an arbitrary modeling language. Therefore, we cannot possibly prove that it is correct in every case. What we could prove here is that the IS_MATCH used for VMTS and VMDL is correct. However, since this would require going into too much technical detail, we choose to omit it here. It is also relatively easy to prove this, as most subtrees in VMDL have a unique identifier (GUID) that we can use during the matching. This remark covers some of the following proofs as well.*

3.1.2 Completeness

We consider that the AST matching phase of a text-based MDM algorithm is complete, if every correctly matched pair appears on its output. In this subsection, we prove that the AST matching phase of our algorithm is complete, assuming some limitations.

Theorem 2. *The AST matching phase of our MDM approach is complete, assuming the matching function IS_MATCH always returns the correct result.*

Proof. Let $T_1 \in AST_1$ and $T_2 \in AST_2$ be two subtrees in the two trees. Let us assume that T_1 and T_2 form a correct pair $(T_1,\ T_2)$ that must be in MP at the end of the phase. T_1 and T_2 can either be on the same level or on different levels of the trees. In the first case, the $MATCH\_ASTS$ subroutine is going to find a match when it is first called from the main algorithm, since we assumed the IS_MATCH operation always returns the correct value. Since it always returns the correct result, T_1 and T_2 can only ever be matched with each other, which means that $\nexists\ T_3 \in AST_2$ so that $(T_1,\ T_3)$ is a match, and $\nexists\ T_4 \in AST_1$ so that $(T_4,\ T_2)$ is a match. Thus, when T_1 and T_2 are not on the same level, they will be unmatched during the first round of matching. However, at the end of the algorithm, we try to match unmatched subtree that we did not try to match before with each other. We have never tried to match T_1 and T_2 before as they were not on the same level during the first round, thus, the algorithm will eventually try to match T_1 with T_2. Therefore, assuming that the IS_MATCH operation is correct, we will always find $(T_1,\ T_2)$ as a pair. □

3.2 Conflict Detection Phase

The conflict detection phase is the second phase of our approach. It uses the result of the AST matching phase to recognize conflicts between two versions of a model. A conflict is an elementary difference between the two versions. It is always related to one or more subtrees in order to accurately track the conflict to model elements. The goal of this phase is to find every conflict and assign solutions to them. A solution is a piece of text that is used to replace the text related to the AST of the conflict. An automatic solution is a solution that is chosen automatically during the merging phase of our method. We differentiate between the following types of conflicts in our approach:

- **Different Text Conflict (DTC).** This type is assigned to a matched subtree pair. A DTC occurs if the raw texts of the matched pair are different. It can either be a semantic (changed attribute, name, etc.) or a non-semantic (comments, etc.) difference. A DTC is recognized by performing a text-based differencing on the pair. A DTC is always assigned to the innermost tree. For example, if an attribute of a node is changed, the conflict is assigned to the subtrees that represent the attributes, instead of the nodes.
- **New Tree Conflict (NTC).** This type is assigned to an unmatched subtree. An NTC occurs if there is a subtree that is present in one version of the model, while it is not present in the other. An NTC is easily recognized as an instance is created

for every unmatched tree found during the AST matching phase.

- **Move Conflict (MC).** This type is assigned to a matched subtree pair. An MC occurs if the positions of the subtrees in a matched pair are not the same. A subtype of this conflict is when the trees in a matched pair are on different levels. This can either have a semantic (i.e. contained node), or a non-semantic (i.e. movement) meaning. An MC is recognized by checking the order of the pair in both trees. If the subtrees are on different levels, it is best recognized by labeling the pair during the matching process.

In previous work, we referred to move conflicts (MC) as order conflicts (OC). We also presented a special subtype when the order of two or three subtrees on the same level were changed. This is a common case in practice, and we deemed it useful for users to easily solve these conflicts with just one solution. However, this is unnecessary on a theoretical level, as the Move Conflict presented here also covers this special case, even though it creates more instances of conflicts in practice. Therefore, in this paper, we exclude this special case from the theoretical presentation.

Algorithm 3 illustrates the conflict detection phase. For every *Unmatched* subtree, the algorithm creates a New Tree Conflict (*NTC*) and adds it to the list. Before doing so, we determine the origin of the subtree (*GET_CONTAINING_TREE*) so we can identify the source of the conflict. For every pair in the *Matched* we found during the AST matching, the algorithm first performs a simple text differencing operation (*TEXT_DIFF*). If the texts of the subtrees differ, then we locate the *INNER_TREES*, so we can identify the source of the conflict. After that, the algorithm adds the created Different Text Conflict (*DTC*) to the list. The next task is checking if the trees were moved. First, we check if the subtrees are on different levels; if they are not, then a Move Conflict (*MC*) is created. If they are on the same level, we have to examine their positions. This is done by checking their related positions to each other in both abstract syntax trees (*DIFF_ORDER*). Checking subtrees that are only present in one AST makes no difference regarding the order of elements. New trees are already handled by another conflict type (*NTC*). If the order of elements is the same in both trees, then there is no conflict. Otherwise, an *MC* is added to the list.

Input: AST_1, AST_2, $Matched$, $Unmatched$
Output: NTC, DTC, MC
for $\forall t \in Unmatched$ **do**
  $AST_R \leftarrow GET\_CONTAINING\_TREE(t)$;
  $NTC.ADD(t, AST_R)$);
end
for $\forall (t_1, t_2) \in Matched$ **do**
  $TDiff \leftarrow TEXT\_DIFF(t_1, t_2)$;
  if $TDiff \neq \emptyset$ **then**
    $(I_1, I_2) \leftarrow INNER\_TREES(AST_1, AST_2, t_1, t_2)$;
    $DTC.ADD(I_1, I_2, TDiff)$;
  end
  if $t_1.LEVEL \neq t_2.LEVEL$ **then**
    $MC.ADD(t_1, t_2)$;
  end
  else
    if $DIFF\_ORDER(t_1, t_2, AST_1, AST_2)$ **then**
      $MC.ADD(t_1, t_2)$;
    end
  end
end

Algorithm 3: Conflict detection - main algorithm.

3.2.1 Soundness and Completeness

We consider that the conflict detection phase of a text-based MDM algorithm is sound, if it does not recognize non-existing conflicts during the recognition. It is complete, if it recognizes every existing conflict between the two differenced versions of the model. In this subsection, we prove that the conflict detection phase of our algorithm is sound and complete. Since the proofs are very similar, we choose to prove them together.

Remark. *Completeness is related to the concept of accuracy. Accuracy defines the ratio of correctly identified conflicts. The conflict detection phase is complete, if it has 100% accuracy. In the case of the more general MDM methods, reaching 100% accuracy is usually a difficult task. Since our method is tailored for VMTS and VMDL, we can reach it more easily. Our future plan is to extend our approach to be more general, so it can be used with other modeling languages as well. This would result in potentially losing this 100% accuracy. However, as we mentioned before, text-based approaches have the advantage on falling back to pure text-based differencing, if - for some reason - the detection algorithm would fail. Therefore, while we might lose the accurate identification of the conflicts, we will never lose completeness, assuming that the textual representations are described correctly.*

Theorem 3. *The conflict detection phase of our MDM approach is sound and complete.*

Proof. There are four places in the algorithm where we recognize a new conflict. Let us examine all these cases individually. The **first case** is the recognition of a New Tree Conflict (NTC) for $\forall u \in Unmatched$. By definition, an NTC is a subtree t so that $t \in AST_1 \cup AST_2$, but $t \notin AST_1 \cap AST_2$. The trees in the $Unmatched$ list cannot be absent from both AST_1 and AST_2, because then the AST matching phase would not have found them. We create every NTC according to the definition, thus, the completeness is proven. We never create an NTC outside of this loop, so the soundness is also proven. The **second case** is the recognition of the Different Text Conflicts (DTC). For $\forall(t1,t2) \in Matched$, the algorithm checks if $Text(t_1)$ and $Text(t_2)$ are different. Raw text-differencing depends on the Longest Common Subsequence problem (Paterson and Dančík, 1994), which is solved for two sequences. Therefore, we can assume that the result of the text differencing is correct. Similarly to the first case, according to the definition, we only add a DTC to the list when $Text(t_1) \neq Text(t_2)$. Therefore, the completeness and soundness are proven. The **third case** is the recognition of a Move Conflict (MC) for $(t_1,t_2) \in Matched$, where $t_1.LEVEL \neq t_2.LEVEL$. This is done according to the definition. The **fourth case** is creating an MC for $(t_t,t_2) \in Matched$ when $t_1.LEVEL = t_2.LEVEL$, but their order is different. We consider checking the relative order of two elements in a list a simple operation and a solved problem. Therefore, we assume that it always returns the correct result. Again, according to the definition, we create a new MC if the order is different in the two trees. Since there are no other cases where we recognize a conflict, and we examined all existing possibilities, the soundness and completeness of the conflict detection phase are proven. □

3.3 Merge Phase

The merge phase of the algorithm is the most practice-oriented phase, because user input heavily influences the outcome of the merging process. The merge phase is split into two phases: 1) the automatic phase, and 2) the iterative phase. During the automatic phase, the automatic solution of every automatically solvable conflict is added to the merged text. In the iterative phase, the user can choose from the designated solutions (i.e. keep the tree or delete the tree), or they can manually solve any conflict with an arbitrary solution. We see no reason to avoid user involvement, as it is expected and conventional in version control systems as a fail-safe method.

Algorithm 4 illustrates the formal description of the merging phase. The inputs of the algorithm are the parsed trees and every recognized conflict. The output is a syntactically and semantically correct merged model given by its textual representation and parsed AST. As we have mentioned before, it is important to have this correctness constraint, since most modeling environments do not support saving incorrect models (Steinberg et al., 2008) (Levendovszky et al., 2005). Therefore, in our opinion, model-based version control systems are correct to apply this constraint as well.

```
Input: AST_1, AST_2, NTC, DTC, MC
Output: Text_M, AST_M
Conflicts ← NTC ∪ DTC ∪ MC;
Auto ← AUTO_CONFLICTS(Conflicts);
Text_M ← GET_TEXT(AST_1);
AST_M ← AST_1;
ASTS ← (AST_1, AST_2);
for ∀c ∈ Auto do
    ITERATIVE_BUILD(Text_M, AST_M, ASTS,
      c, Conflicts, ∅);
end
while ¬Done do
    Command ← USER_INPUT();
    if Command == (c_u, Solution) then
        ITERATIVE_BUILD(Text_M, AST_M,
          ASTS, c_u, Conflicts, Solution);
    end
    if Command == Finalize then
        if CHECK(Text_M, AST_M) then
            Done = true;
        end
    end
end
```

Algorithm 4: Merging - main algorithm.

First, the algorithm takes one of the texts ($GET\_TEXT(AST_1)$) as basis for the merging . It then continuously tries to build up the merged text $Text_M$ and the merged tree AST_M. In the automatic phase, the algorithm takes every automatically resolvable conflict ($AUTO\_CONFLICTS$), and builds the merged tree iteratively ($ITERATIVE\_BUILD$). It is important to note that the iteratively built merged tree may not always be correct. It is the responsibility of the user to ensure that the end result is a correct model given in its textual form. Thus, we have to keep track of every conflict by its absolute position in the merged text, instead of its position in the merged AST. Algorithm 5 illustrates the iterative building process that is used during both the automatic and the iterative phases. It first resolves the conflict in the text automatically ($AUTO\_RESOLVE$) or by using a given solution ($RESOLVE$). Then, it builds the merged tree ($BUILD\_TREE$) based on

Input: $Text_M$, AST_M, $ASTS$, c, $Conflicts$, $Solution$
Output: $Text_M$, AST_M
if $Solution == \emptyset$ **then**
  $Text_M \leftarrow AUTO\_RESOLVE(Text_M, c)$;
end
else
  $Text_M \leftarrow RESOLVE(Text_M, c, Solution)$;
end
$AST_M \leftarrow$
 $BUILD\_TREE(AST_M, ASTS, Text_M)$;
$I \leftarrow GET\_INTERACTIONS(Conflicts, c)$;
$UPDATE\_POSITIONS(AST_M, Conflicts, I)$;

Algorithm 5: The ITERATIVE_BUILD operation.

the change that occurred. Afterwards, we have to update the positions of the conflicts by their position in the text (*UPDATE_POSITIONS*). The *GET_INTERACTIONS* operation discovers the interactions the conflicts have on each other. In the iterative phase, the user has complete control over the resolution of the remaining conflicts (*USER_INPUT*). For every resolved conflict, we call the *ITERATIVE_BUILD* operation with the *Solution* that the user chose. The iterative building is the same as it was in the automatic phase. Finally, when the merging process is over, we have to *CHECK* if the merged model is correct. Similarly to the *IS_MATCH* function presented in Section 3.1, the *CHECK* operation uses the parser of the language to build a model from the text, and then checks the correctness of the model.

3.3.1 Soundness

We consider that the merge phase of a text-based MDM algorithm is sound, if it cannot produce an incorrect merged model on its output. In this subsection, we prove that the merge phase of our algorithm is sound, assuming the checking function *CHECK* always returns the correct result.

Theorem 4. *The merge phase of our MDM approach is sound, assuming the checking function CHECK always returns the correct result.*

Proof. The iterative phase of the algorithm ends ($\neg$*Done*) once we verify that the model is both syntactically and semantically correct. This is done by the *CHECK* operation, which is similar to the *IS_MATCH* operation in the AST matching phase. Both are configurable operations that rely on the parser of the language (VMDL in our case). Since we assumed that the *CHECK* operation is correct, the merging phase can only end with a correct model. The assumption is reasonable for the same reasons we have seen before in Section 3.1.1. □

3.3.2 Completeness

We consider that the merge phase of a text-based MDM algorithm is complete, if the merged model contains a solution for every discovered conflict. In this subsection, we prove that the merge phase of our algorithm is complete, assuming it is sound.

Theorem 5. *The merge phase of our MDM approach is complete, assuming it is sound.*

Proof. The algorithm takes one version of the model (AST_1 and $GET\_TEXT(AST_1)$), and proceeds to build the merged model based on this version. In Section 3.1.1, we proved that the end result of the merge phase is always a syntactically and semantically correct model. Therefore, the merged model must contain a solution for conflicts that would make the model incorrect. During the iterative phase, the user must give a solution for these conflicts. All that we have to prove is that the conflicts that do not cause the model to be incorrect have a solution in the merged model. Conflicts with an automatic solution are solved during the automatic phase. For the rest of the conflicts, let us examine them by type. The New Tree Conflicts are either present or not in AST_M, based on their appearance in AST_1. In the case of the Different Text Conflicts, the text that is used in the merged model appears in $Text_M$. For the Move Conflicts, the solution is the order found in AST_1, which is also always present. These solutions cannot cause the merged model to be incorrect, because the *Check* operation at the end of the algorithm would fail. We have seen that every conflict has a solution in the merged model, thus, the merge phase of our algorithm is complete. □

4 CONCLUSIONS

In this paper, we have verified our previously presented text-based model differencing and merging (MDM) method according to different aspects. We have also formally described the algorithm. The formal description was not presented in previous work. The algorithm is capable of differencing and merging two versions of a model by their textual representations. We have briefly introduced the modeling framework (VMTS) we are using and the textual language (VMDL) that is used to describe the textual representations of the models. We have also discussed the differences between traditional text differencing and merging, and text-based MDM approaches. We have

concluded that text-based MDM methods are needed for the differencing and merging of the textual representations.

Our text-based MDM approach consists of three phases: 1) the abstract syntax tree (AST) matching phase, 2) the conflict detection phase, and 3) the merging phase. We presented the formal description of our algorithm with pseudo code, divided by the three phases of the algorithm. Afterwards, we verified the algorithm based on two aspects: soundness and completeness. We proved that during the AST matching phase, the algorithm does not find any incorrectly matched pairs, and that it always finds every correct pair. We also verified that during the conflict detection phase, the algorithm does not recognize any incorrect conflicts; and that it recognizes every existing conflict. Finally, we proved that during the merging phase, it is not possible to create an incorrect merged model, and that every conflict would be solved in some way in the resulting merged model.

We have also discussed the differences between text-based and graph-based MDM approaches, and outlined our motivation for researching text-based MDM approaches. The most significant of these were the following: better support for serialization, a failsafe during the differencing process, and the fact that there are few such existing methods available. We have also mentioned some areas where text-based MDM methods can be applied, such as serialization and version control, or during the synchronization of the textual and graphical notations.

4.1 Future Work

Based on the research presented in this paper, we consider the following to be promising directions for future work:

- **Comparison with structure-based approaches.** Since there are few existing text-based MDM approaches, we are interested in seeing how our method compares to structure-based approaches regarding characteristics like accuracy, performance, or generality. The next step would be comparing the two types of approaches on a more general and theoretical level.
- **Complexity analysis.** In future work, we plan to extend the verification of our approach by examining its complexity. We plan to analyze the worst-case complexity of the different phases, and compare these to the complexity of basic algorithms that can be used to solve these problems. The basic algorithm for the AST matching phase would be matching every subtree in one AST with every subtree in the other AST. We aim to prove that the worst-case complexity is on par with these algorithms, but in practice, our method usually performs better due to the heuristics we use.
- **Extend the method to be more general.** Our original goal was to create a text-based MDM method that is able to handle arbitrary VMTS models described by VMDL. Our goal for the future is to extend this method by making it more general, meaning that it should be usable with other modeling languages as well. During the development of the algorithm, we have kept this goal in mind. Some parts of the algorithm (like the aforementioned *IS_MATCH* or *CHECK* operations) are already created with this in mind. Having a general text-based MDM method would benefit the text-based modeling research field, as there are very few such existing methods available. The long-term use-case for this would be creating a text-based version control system for models that is not dependent on any modeling environment.

ACKNOWLEDGEMENTS

This work was performed in the frame of FIEK_16-1-2016-0007 project, implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the FIEK_16 funding scheme.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. (2005). *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc.

Altmanninger, K., Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., and Wimmer, M. (2009). Why Model Versioning Research is Needed!? An Experience Report. In *Proc. of the MoDSE-MCCM 2009 Workshop @ MoDELS 2009*.

Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., and Varró, D. (2015). Viatra 3 : A reactive model transformation platform. In *8th International Conference on Model Transformations*, L'Aquila, Italy. Springer, Springer.

Brun, C. and Pierantonio, A. (2008). Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34.

Eysholdt, M. and Behrens, H. (2010). Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proc. of the ACM International Conference*

Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '10, pages 307–309, New York, NY, USA. ACM.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 313–324, Västeras, Sweden.

Gall, H. C., Fluri, B., and Pinzger, M. (2009). Change analysis with evolizer and changedistiller. *IEEE Software*, 26(1):26–33.

Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., and Völkel, S. (2007). Text-based Modeling. In *Proc. of the 4th International Workshop on Software Language Engineering*.

KDiff3 (2003). A text-based differencing and merging tool. http://kdiff3.sourceforge.net/.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr.

Levendovszky, T., Lengyel, L., Mezei, G., and Charaf, H. (2005). A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. *Electronic Notes in Theoretical Computer Science*, 127(1):65–75.

Lin, Y., Zhang, J., and Gray, J. (2004). Model comparison: A key challenge for transformation testing and version control in model driven software development. In *Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development*, pages 219–236. Springer.

Meta Object Facility (2003). Meta object facility (MOF) 2.0 core specification. Version 2.

Moher, T. G., Mak, D., Blumenthal, B., and Levanthal, L. (1993). Comparing the comprehensibility of textual and graphical programs. In *Empirical Studies of Programmers: Fifth Workshop*, pages 137–161. Ablex, Norwood, NJ.

Paige, R. F., Matragkas, N., and Rose, L. M. (2016). Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272 – 280.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition.

Paterson, M. and Dančík, V. (1994). *Longest common subsequences*, pages 127–142. Springer Berlin Heidelberg, Berlin, Heidelberg.

Pérez Andrés, F., de Lara, J., and Guerra, E. (2008). *Domain Specific Languages with Graphical and Textual Views*, pages 82–97. Springer Berlin Heidelberg, Berlin, Heidelberg.

Petre, M. (1995). Why looking isn't always seeing: Readership skills and graphical programming. *CCommunications of the ACM*, 38(6):33–44.

Sendall, S. and Kozaczynski, W. (2003). Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45.

Somogyi, F. and Asztalos, M. (2016). Merging textual representations of software models a practical approach. In Hnatkowska, B. et al., editors, *Software Engineering: Improving Practice through Research*. Polish Information Processing Society.

Somogyi, F. A. (2016). Merging Textual Representations of Software Models. In *MultiScience - 2016. microCAD International Multidisciplinary Scientific Conference*, Miskolc, Hungary.

Spinellis, D. (2005). Version control systems. *IEEE Software*, 22(5):108–109.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.

van Rest, O., Wachsmuth, G., Steel, J., Süß, J. G., and Visser, E. (2013). Robust Real-Time Synchronization between Textual and Graphical Editors. In *Theory and Practice of Model Transformations, Sixth International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings.*, Lecture Notes in Computer Science. Springer Verlag.

van Rozen, R. and van der Storm, T. (2015). *Origin Tracking + Text Differencing = Textual Model Differencing*, pages 18–33. Springer International Publishing, Cham.

VMTS (2003). Visual modeling and transformation system. http://vmts.aut.bme.hu.

XML Metadata Interchange (2015). Xml metadata interchange (XMI) specification. Version 2.5.1.