Detecting Renamings in Three-Way Merging

László Angyal, László Lengyel, Hassan Charaf

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
{angyal, lengyel, hassan}@aut.bme.hu

Abstract: Teamwork is the typical characteristic of software development, because the tasks can be splitted and parallelized. The independently working developers use Software Configuration Management (SCM) systems to apply version control to their files and to keep them consistent. Several SCM systems allow working on the same files concurrently, and attempt to auto-merge the files in order to facilitate the reconciliation of the parallel modifications. The merge should produce syntactically and semantically correct source code files, therefore, developers are often involved into the resolution of the conflicts. However, when a general textual-based approach reports a successful merge, the output can still be failed in compile time, because semantic correctness cannot be ensured trivially. Renaming an identifier consists of many changes, and can cause semantic errors in the output of the merge, which subsequently have to be corrected manually. This paper introduces that matching the identifier declarations, e.g. class, field, method, local variables, with their corresponding references in the abstract syntax trees of the revisions, and considering the detected renamings during the merge takes closer to semantic correctness. The problem is illustrated and a solution is elaborated in this work.

Keywords: Three-way Merge, Abstract Syntax Tree, Refactoring, Renaming Identifiers, Semantic Errors

1 Introduction

There are two traditional concurrency models among the source code management (SCM) systems: lock and merge models. The lock model prevents the concurrent modification on the same files, but the merge model allows the parallel editing, and performs a merge to reconcile the changes. A three-way merge engine is a usual part of SCM systems, and some of them attempt to auto-merge the files, but they often fail due to textual-based approaches or semantic conflicts. The best methods should treat modifications as semantic changes in high abstraction level, rather than atomic changes. The atomic changes do not reflect the intentions of the developers at all, therefore, discovering those intentions can significantly improve merge approaches.

Refactoring [1] means restructuring the code of a system without modifying its run-time behaviour. Refactorings are composite changes in higher abstraction level. In contrast to simple low-level atomic changes, they aim at improving several characteristics of the software source code e.g. understandability, maintainability. For instance, renaming an identifier to a better name, can help the understability, while this renaming activity consist of numerous atomic changes in the code.

The reconcilation of two modified revisions of a source code file is referred to as merge. Merge includes (i) finding the last incorporated changes by differencing, (ii) conflict detection and resolution, and finally (iii) the propagation of changes in order to produce the reconciled version. The differencing matches the corresponding elements (e.g. lines, syntax elements) of the original and the altered file. The modifications are derived from the non-matched elements. The output of the differencing is the list of edit operations that are applied to the original file provides the modified one. The three-way merge approach (illustrated in [2]) is an unambiguous way of detecting the modifications in the altered revisions, and where the original file is also taken into account. Change propagation is performed by replaying the detected edit operations.

The granularity of the merge means the size of the smallest indivisible changes that can be detected and then propagated. Obviously, the fine-grained methods have slower execution time over coarse-grained ones, but better conflict resolution can be achived by a fine-grained merge. For example, the widespread line-based textual algorithm [3] detects even the smallest change as the line changed. More changes within the same line became invisible and the source of further merge conflicts. Usually, there are relations among the independent changes, which involve certain semantic meanings as well. These relations should be considered while merging revisions of files. The refactorings affect many lines of the file. The typical merge engines handle the composite changes as set of independent atomic changes. This makes them unfeasible for merging files after refactoring.

Fine-grained approaches like abstract syntax tree (AST)-based approaches (e.g. [4], [5]) are more suitable for source code differencing and merging, because, with the knowledge of the language's syntax, they always produce syntactically correct output contrary to line-based textual approaches e.g. the diff3 tool [6]. However, semantic correctness is not ensured at all by considering the syntax. Furthermore, after the merge, the reconciled AST has to be serialized. The technique that visits all the nodes of the tree to emits source code is called pretty-printing [7]. An AST-based merge is language dependent and works with lower performance, therefore, it is rarely used in general versioning systems.

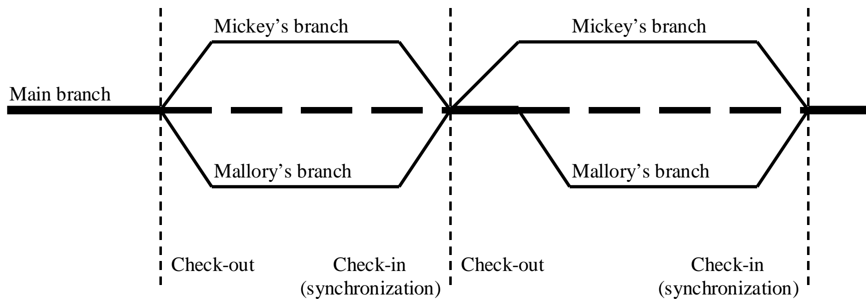

Figure 1
Developing process of the running example

Consider the evolution of a software, which is developed by two users, Mickey and Mallory. They use a versioning system that supports the merge concurrency model. A part of the development process can be followed in Figure 1. After checking-out the files, both of them can modify the same files. Mickey renames a class without telling Malory to do the same. Mallory uses a reference to that class in her inserted lines. After a successful merge performed by the versioning system, they found that the merged file contains some semantic errors. The inserted class references were not corrected with the new name of that class. This paper discusses a solution for these problems.

The renamings should be detected before the merge and should be considered while reconciling the changes. The identifier declarations and their corresponding reference nodes have to be collected by traversing the ASTs of the files. Using the matches from the differencing, a mapping has to be found between the identifiers in different ASTs, which can be used to the correction of the renamed identifiers.

The remainder of the paper is organized as follows. We discuss the semantic conflict related to identifier renaming in Section 2. A solution for that problem is described in Section 3. This is followed by the introduction of the existing approaches and tools, and finally conclusions and future work are elaborated.

2 Problem Statement

Consider the situation, where the developers checked-out a source file to modify independently at the same time. The used file versioning system performs an AST-based three-way merge on the concurrently altered files after file check-ins. Figure 2 depicts the original file and both modified revisions by Mickey and Mallory as well. The merge first compares the revisions to the original file, to detect changes. The difference analysis of the Mickey's version produces the following differences as atomic AST node operations.

```
using System.Drawing;

public interface Widget {
   void Paint(Graphics g);
   void SetLocation(Point p);
   void SetSize(Size s);
}

public class Label : Widget {

   string label;
   Point location;
   Size size;

   public void Paint(Graphics g) {
      g.DrawString(this.label, this.location);
   }

   public void SetLocation(Point p) {
      this.location = p;
   }

   …

   public override string ToString() {
      string str;
      str = this.label;
      return str;
   }
}
```

Original file

Mickey's revision

```
using System.Drawing;

public interface Control {
   …
}

public class StaticText : Control {
   …
   public void Paint(Graphics
graph) {
      graph.DrawString(…);
   }

   …

   public override string
ToString() {
      string value;
      value = this.label;
      return value;
   }
}
```

Mallory's revision

```
using System.Drawing;

   …

public class Label : Widget {
   …
   public override string
ToString() {
      string str;
      str = this.label;
      return "[" + str + "]";
   }
}

public class Button : Widget {
   Label label;

   public void Paint(Graphics g) {
      g.DrawRectangle(…);
      this.label.Paint(g);
   }
   …
}
```

Figure 2

The original and the modified files

OP	Name	Type of the AST node	Name of the parent node	New value
UPD	*Widget*	TypeDeclaration	Global_Types	Control
UPD	*Label*	TypeDeclaration	Global_Types	StaticText
UPD	*g*	ParameterDeclarationExpression	Paint_Parameters	graph
UPD	*g*	VariableReferenceExpression	DrawString	graph
UPD	*str*	VariableDeclarationStatement	CodeStatementCollection	value
UPD	*str*	VariableReferenceExpression	Assign (str=label)	value
UPD	*str*	VariableReferenceExpression	return str	value

Table 1

Mickey's version contains some updates

There are relations among the identified edit operations (Table 1): (i) parameter declaration of *g* has been changed to *graph*, and consequently, the reference to *g* has also been changed to *graph*, (ii) local variable *str* has been changed to *value* and their corresponding references as well. From the high abstraction level semantical point of view, these are two composite changes, not a list of independent atomic changes.

Mallory has not updated anything existing (Table 2), but she has inserted a new class *Button* and reused the existing interface *Widget* and the class *Label*. She has changed an expression with a previously declared local variable *str*.

OP	Name	Type of the AST node	Index	Name of the parent
INS	*Button*	TypeDeclaration	2	Global_Types
INS	*label_0_Label*	MemberField	0	Button
INS	*Paint_Public_Graphics*	MemberMethod	1	Button
INS	*Add*	BinaryOperatorExpression	0	return
INS	*Add*	BinaryOperatorExpression	0	Add (left side)
INS	*"["*	PrimitiveExpression	0	Add (left side)
INS	*"] "*	PrimitiveExpression	1	Add (right side)
MOV	*str*	VariableReferenceExpr	1	Add (right side)
…				

Table 2

Mallory's version contains inserts

The merge does not detect any conflicts between the two list of operations. Executing the edit operations of Mickey's file on Mallory's version, without any semantic considerations, produces the output depicted in Figure 3. The file is still syntactically correct, but contains several semantic errors, which have to be corrected manually after the merge.

```
using System.Drawing;

public interface Control {
   …
}

public class StaticText : Control {
   …

   public override string ToString() {
       string value;
       value = this.label;
       return ("[" + str + "]");
   }
}

public class Button : Widget {
   Label label;

   public void Paint(Graphics g) {
       g.DrawRectangle(this.location, this.size);
       this.label.Paint(g);
   }
   …
}
```

Figure 3

Merged version with certain semantic errors

The variable *str* has been renamed to *value*, class *Label* has been renamed to *StaticText* and interface *Widget* to *Control*, according to the edit operations. However, the newly inserted reference to *str* remains *str* and the class *Button* tries to implement the already renamed interface *Widget*. A merge relies only on the detected edit operations and replays them without sense, this can easily produce compile time errors. The merge should correct the errors by detecting the renames and applying the new names in the newly inserted references.

3 A Renaming-Aware Extension

The purpose of this extension is to extend a three-way merge approach with the ability to be renaming-aware. When reconciling two source files with a renaming that has been performed in one of them, then the newly inserted references with the old identifier names must be renamed as well in order to ensure the semantic correctness. Previous section has shown that merge engines should take the identifier renaming into account and this section proposes a solution, that is illustrated via AST nodes of Microsoft's .NET i.e. CodeDOM [8] nodes.

The two major points of our approach are

(i) discovering the identifier dependencies and building a lookup table of the identifier declarations and the corresponding references with fully qualified names,

(ii) while executing the edit operations, the identifier dependencies are taken into account.

Before describing point (i), we take a closer look at the different types of identifier declaration nodes and their dependencies.

Declaration node	Place of the declaration	References node
VariableDeclaration	In method bodies with unique name	*VariableReferenceExpression*
ParameterVariable-Declaration	In method signatures: method parameter block	*VariableReferenceExpression*

Table 3

Local variable declaration nodes

The union of the visibility scope of local variables with the same name is prohibited within a method body, and a variable (Table 3) with the name of a parameter variable in the method signature cannot be declared, since Java or C# compilers report error. A global lookup table with fully qualified variable names is enough to unambiguously select a certain identifier declaration node.

Declaration node	References nodes
Namespace	In fully qualified *TypeReference* or *VariableReferenceExpression*
Class/Structure *TypeDeclaration*	Base class in class declaration (*TypeReferenceExpression*) Static method invocation (*VariableReferenceExpression*) Static field reference (*VariableReferenceExpression*) Field type (*TypeReference*) Variable type (*TypeReference*) Object creation (*ObjectCreateExpression*) Array type (*ArrayCreateExpression*) Casting (*CastExpression*) Generics (*TypeReference*)
MemberField	*FieldReferenceExpression*
MemberMethod	Method invocation (*MethodReferenceExpression*)
MemberEvent	*EventReferenceExpression*
MemberProperty	*PropertyReferenceExpression*

Table 4

Identifiers with global visibility

The full name comprises the namespace, the name of the class, the method that contains that local declaration, and finally, the variable name as well as the order of its declaration if there are more variables with the same name within a method.

Table 4 summarizes the identifiers with global visibility beside some possible reference nodes that are offered by CodeDOM.

Figure 4 illustrates the partial AST of the running example with its lookup table, and the relations between the nodes and the rows in the table. The identifier lookup table contains the identifiers with their fully qualified names, the link to the declaration node (red arrows), and the list of links to the correcponding reference nodes (blue arrows) as well. This lookup table can be built by traversing the AST before the merge.

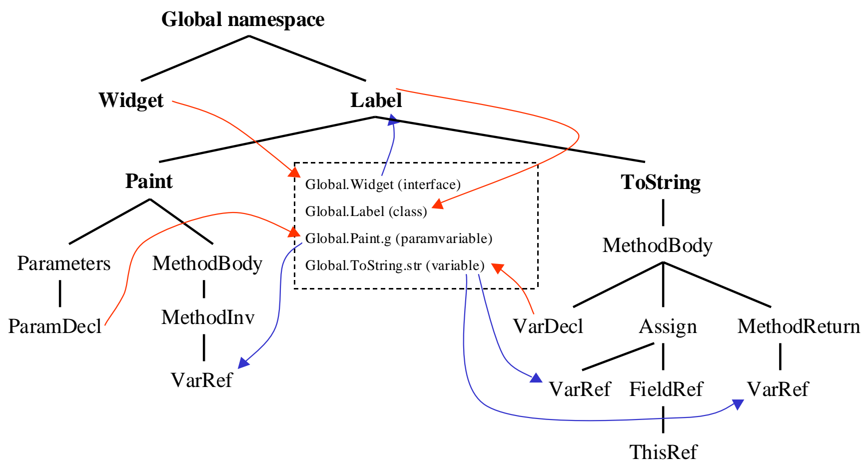

Figure 4
Partial AST of the original version of the code in the running example and its identifier lookup table

According to item (ii), identifier dependencies are considered while doing the merge. We distinguish between two kinds of operation: (a) insert a new node and (b) update an existing node.

First of all, we need a mapping between the lookup tables of the different ASTs. The common point of these different tables is that the difference analysis of the two ASTs matches the corresponding nodes in different trees. For instance, in Figure 4 variable declaration node *str* is matched with variable declaration node *value* in Figure 5, thus, even if their fully qualified name is different, there is a mapping between these nodes. This aim needs a differencing with an identifier name independent match.

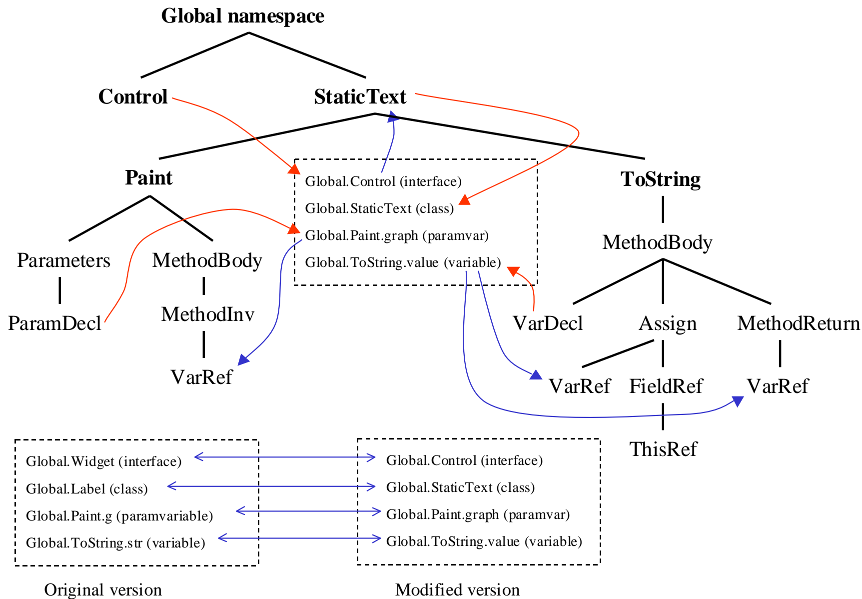

Figure 5

Mickey's version and the mapping between lookup tables

In case (a) when inserting a new reference node, the dependency table should be looked at. If the reference name differs from its declaration name, then the reference name that is going to be inserted must be renamed. Figure 3 illustrates that a local variable *str* is inserted without checking its declaration name, which has been changed to *value* meanwhile, due to Mickey's work. The mapping between the tables allows to look up the matching between the declaration node *str* and the declaration node *value*. Along these connections, we can found that the declaration name is different. Therefore, the algorithm should rename the new variable reference node that is to be inserted, and the new name has to be *value*.

In case (b), when updating an indentifier declaration node, the corresponding references, which also store the name of the identifier have to be changed as well. These reference nodes can be looked up from the table. For example, if we want to execute the edit operations from Mickey's version on Mallory's file, updating the interface declaration *Widget* to *Control* should involve the change of the class reference in the declaration of *Button* from *Widget* to *Control*.

4 Existing Tools and Approaches

This section introduces some of the most relevant tools and approaches that are related to our work.

The well-known *CVS* [9] uses line-based textual merge. Due to its coarse-grained granularity, it detect atomic changes together with their context, for instance, renaming a variable in an expression indicated as the whole line has been changed. After a successful merge of files that were edited in parallel, syntactical and semantical errors can remain in the source code. These problems must be corrected manually after the merge, which was reported to be finished without conflicts. The errors that are revealed in compile time are better than run-time errors, because they are hidden e.g. unintended method overrides and can cause the malfunction of the software.

A common characteristic of textual and AST-based differencing is that they detect several atomic changes without connection between them, abstraction of the changes should be extracted to guess the intentions of the developer. [10] presented that identifying the relations of the atomic changes is important to improve the comprehension of the source code evolution. Small changes can be grouped together into high-level abstract operations. Other advantage of the abstraction is that the changes become reusable on other files.

The state-of-the-art approaches handle source code changes as semantic actions, because they present more information and reflect the intentions of the developers. The differencing techniques that detect changes in lines or in ASTs provide the list of atomic changes e.g. insertion or deletion of a node, but these changes have no abstract information value. The modern integrated development environments (IDE) have the ability to log the semantic changes in high abstraction level and the corresponding low-level details as well. For instance, *Eclipse* [11] has a refactoring engine that logs the changes performed by refactoring actions, like renaming a variable or a class, if they were done via that engine. These logs can be utilized further during the merge process.

Molhado is a refactoring-aware SCM system, that includes an *Eclipse* plugin *MolhadoRef* [12], which captures and stores the performed refactorings on Java files. Its underlying data model is flexible and allows representing programs in any language. It performs only lightweight parsing due to performance reasons, the method bodies in string format are handled as attributes of methods. *MolhadoRef* use the *Eclipse* built-in differencer engine to perform textual difference analysis, the changed lines are examined if the changes were caused by the refactoring operations, and if so, they are removed from the change list. After that, *Molhado* can perform a textual merge by replaying the recorded refactorings together with other edit operations in order to propagate changes. Authors of *MolhadoRef* exhibits better merge results with less human intervention compared to *CVS*.

The merge based on previously saved logs is called operation-based merging. These logs recorded by the source code editor contain text edit operations and informations on other high level source code transformations. Operation-based approaches can be very precise in recording the changes and replaying them, but this technique heavily relies on the editor, and sometimes the log files are unavailable or inconsistent with the changes, because it cannot be ensured that every developer uses the predefined editor. *RefactoringCrawler* [13] is a tool that can reconstruct with good reliability some kinds of applied refactorings by comparing the original and the modified version of a Java file. It uses user adjustable parameters to match the method bodies of the classes. Its matching algorithm is based on an approach that uses fingerprints of the tokenized method bodies. After matching, it performs semantic analysis. *RefactoringCrawler* is limited to examine API interfaces, it does not deal with local variables and has some shortcomings with fields.

In [14], a tool is presented that detects and reports the name and type changes in identifiers of different versions of a C program. The purpose of this tool is to improve the understanding of software evolution with higher level abstract information about the name and type changes. It uses a *TypeMap* for typedefs, structures and unions, a *GlobalNameMap* for global variables, and *LocalNameMap*s per function bodies to collect the matched identifiers. Types and functions are matched if they have the same name. The AST traversal within function bodies is performed by parallel and the local variables are mapped by their syntactical position.

In our approach, there is AST-based differencing and execution of atomic node operations, where the related identifier declarations and references are connected together and taken into consideration while applying those detected operations on the other AST. If any of the identifier nodes are changed, it should also affect the others. If the name of a variable is changed in the declaration, we modify every references that have to be refreshed. If the code, which is taken as input is semantically correct i.e. can be compiled, the identifier references must be consistent with their corresponding declarations. The advantage of our method over other object-oriented tools is that we support local variables.

Conclusions and Future Work

The importance of the comprehension of the committed changes has been pointed out via an illustrative example. The detection of the renamings should be considered in any three-way merge systems, to avoid tedious and error-prone manual code reviews and fixing the code after merging. The presented approach takes us closer to a semantically correct merge. Although, there are several other semantic related problems that were not addressed, however, huge number of compile-time errors can be reduced by the presented approach, and it moves toward an automatic merge without human interaction. Our future work involves the reseach of the solutions to other semantic problems.

The approach can work in merging generated code with manually written code, where refactorings are not explicitly intended by the developers, but caused by the code generator, because some parameters have changed. As future work, we also plan to improve the presented approach to create an efficient code generation tool with round-trip engineering support. The tool can be used in a visual designing environment, which applies bi-directional validated model to source transformations. Round-trip engineering of custom models lacks tool support due to the complexity and the difficulty of the generalization of all specific cases.

Acknowledgement

The fund of 'Mobile Innovation Centre' has partly supported the activities described in this paper. This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

[1] Martin Fowler et al., Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999, ISBN 0201485672

[2] Tom Mens, A State-of-the-Art Survey on Software Merging, IEEE Transactions on Software Engineering, 28(5), May 2002, pp. 449-462

[3] Eugene W. Myers, An O(ND) Difference Algorithm and its Variations, Algorithmica, 1(2), 1986, pp. 251-266

[4] Wuu Yang, How to Merge Program Texts, Journal of Systems and Software, Vol. 27, No. 2, 1994, pp. 129-135

[5] Ulf Asklund, Identifying Conflicts During Structural Merge, Proceeding of the Nordic Workshop on Programming Environment Research '94, Lund University, 1994, pp. 231-242

[6] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A Formal Investigation of Diff3, Manuscript, University of Pennsylvania, 2006

[7] Derek C. Oppen, Prettyprinting ACM Transactions on Programming Languages and Systems, 2(4), 1980, pp. 465-483

[8] Microsoft's CodeDOM Web Site, http://msdn2.microsoft.com/en-us/library/ system.codedom.aspx

[9] CVS Wikipedia Web Site, http://ximbiot.com/cvs/wiki/

[10] Peter Ebraert, Jorge Antonio Vallejos Vargas, Pascal Costanza, Ellen Van Paesschen, Theo D'Hondt, Change-Oriented Software Engineering, 15th International Smalltalk Joint Conference, Lugano, Switzerland (to be published), 2007

[11] Eclipse Web Site, http://www.eclipse.org

[12] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen, Refactoring-Aware Configuration Management for Object-Oriented Programs. International Conference on Software Engineering. IEEE Computer Society, Washington, DC, pp. 427-436

[13] Danny Dig, Can Comertoglu, Darko Marinov, Ralph Johnson, Automated Detection of refactorings in evolving components, European Conference on Object-Oriented Programming, Nantes, France, 2006, pp. 404-428

[14] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks, Understanding source code evolution using abstract syntax tree matching. ACM SIGSOFT Software Engineering Notes 30(4), July 2005, pp. 1-5