Magyar Kutatók 8. Nemzetközi Szimpóziuma
8th International Symposium of Hungarian Researchers on Computational Intelligence and Informatics

Object-Oriented Identifier Renaming Correction in Three-Way Merge

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: 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, some of them attempt to auto-merge the files, but sometimes they fail due to textual-based approaches or semantic conflicts. The merge should produce syntactically correct source files, but semantic correctness cannot be ensured trivially. The best methods treat modifications as semantic changes in high abstraction level, rather than atomic changes. The atomic changes do not reflect the intentions of the developers, therefore discovering those intentions can significantly improve semantic merge approaches. This paper introduces that matching the corresponding identifiers e.g. class, field, method, local variables in the ASTs of the revisions, and detection of renaming takes closer to semantic correctness. Renaming of an identifier can cause semantic errors in the output of the merge. This issue is examined and a solution is elaborated in this work.

Keywords: Three-way Merge, SCM, Refactoring, Renaming Identifiers, Semantic Errors

1 Introduction

Refactoring [1] means restructuring the code of an object-oriented 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 the goal to improve several characteristics of the software source code e.g. understandability, maintainability. For instance, renaming an identifier to a better name helps the understability, while the renaming activity causes numerous atomic changes in the code.

Refactorings affect many nodes of the abstract syntax tree (AST). Changes are reconciled by detecting the changed nodes and edit operations are constructed that can propagate those changes to the other side. The typical merge engines handle the composite changes as set of independent atomic changes. This makes them unfeasible for merging files after refactoring.

The granularity of the merge means the size of the smallest indivisible changes that can be 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, a line-based textual approach detects even the smallest change as the line changed. More changes within the same line became invisible and source of further merge conflicts. Usually, there are relations among the independent changes, which involves some semantic meanings as well. These relations should be considered while merging revisions of files.

AST-based merge approaches are more suitable for source code differencing and merging, because they always produce syntactically correct output contrary to line-based textual approaches e.g. diff, but semantic correctness is not ensured at all. An AST-based merge is language dependent and works with lower performance, therefore, it is rarely used in general versioning systems.

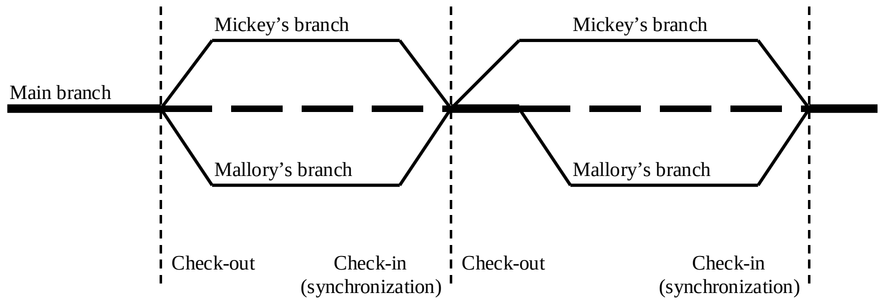

Figure 1
Developoing process of the running example

Assume 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. The whole process of development can be followed in Figure 1. After checking-out the files and 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 successfull 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 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

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

Table 1
Mickey's version contains some updates

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	*str*	VariableDeclarationStatement	CodeStatementCollection	value
UPD	*str*	VariableReferenceExpression	Assign: "str=label"	value
UPD	*str*	VariableReferenceExpression	return str	value
UPD	*g*	VariableReferenceExpression	DrawString	graph

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

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

Table 2
Mallory's version contains inserts

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)
…				

```
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
Original and modified files

The merge replays the edit operations without any semantic consideration and produces the output depicted in Figure 3. The output 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 some semantic errors

The variable *str* has been renamed to *value*, class *Label* was renamed to *StaticText* and interface *Widget* to *Control* according to the edit operations. However, the newly inserted reference to *str* remained *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 The Renaming-Aware Extension Approach

The purpose of this approach is to extend a three-way merge approach with the ability to be renaming-aware. When reconciling two source files, while renaming was performed in one of them, then the newly inserted references with the old identifier names have to be renamed as well, in order to ensure the semantic correctness. Previous section has showed that merge engines should take the identifier renaming into account and this section proposes a solution, that is illustrated via .NET AST i.e. CodeDOM [2] nodes.

The two main 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 edit operations, the identifier dependencies are taken into account.

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

Table 3
Local variable declaration nodes

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*

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, because the full name comprises the namespace, the name of the class, the method that contains that local declaration and finally, the variable name, and the order of its declaration if there are more variables with the same name within a method. In Table 4 we summarize the identifiers with global visibility beside some possible reference nodes that are offered by CodeDOM.

Table 4
Identifiers with global visibility

Declaration node	References nodes
Namespace	In fully qualified *TypeReference* or *VariableReferenceExpression*
Class/Structure: *TypeDeclaration*	Baseclass 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*

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 symbol lookup table contains the identifiers with their fully qualified names. The lookup table can be built by visiting the AST nodes of the parsed code.

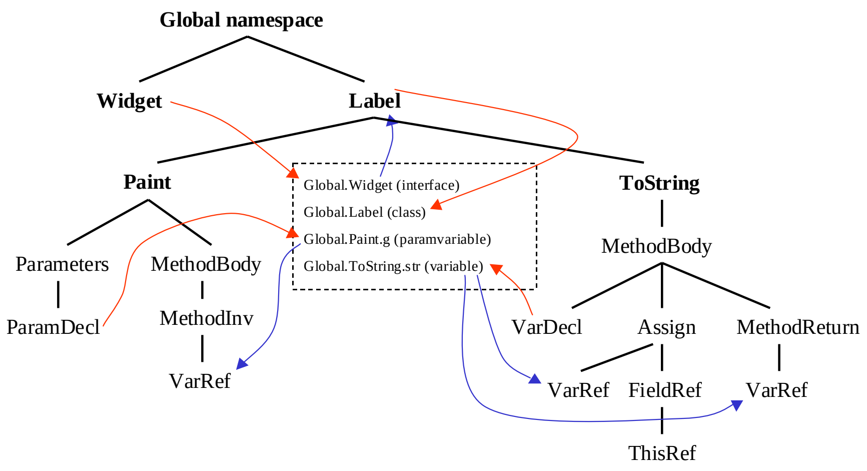

Figure 4
Partial AST of the original version of the code in the running example and its identifier lookup table

According to point (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 of these different tables is that, the differencing 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.

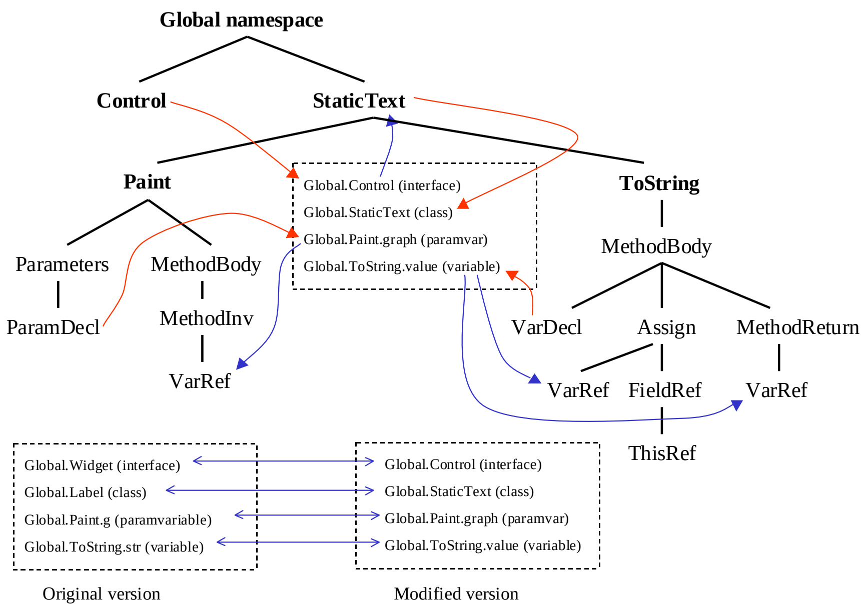
Global namespace
Control
StaticText
Paint
ToString
Parameters
MethodBody
ParamDecl
MethodInv
VarRef
Global.Control (interface)
Global.StaticText (class)
Global.Paint.graph (paramvar)
Global.ToString.value (variable)
MethodBody
VarDecl
Assign
MethodReturn
VarRef
FieldRef
VarRef
ThisRef
Global.Widget (interface)
Global.Label (class)
Global.Paint.g (paramvariable)
Global.ToString.str (variable)
Global.Control (interface)
Global.StaticText (class)
Global.Paint.graph (paramvar)
Global.ToString.value (variable)
Original version
Modified version

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 changed to that name. Figure 3 illustrates that a local variable *str* is inserted without checking its declaration name, which was meanwhile changed to *value*, 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 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 interface declaration *Widget* to *Control* should involve the changing of the class reference in 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* 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 was changed. After a successful merge of files that were edited in parallel, syntactic and semantic errors can remain in the source code. 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 a lot of atomic changes without connection between them, abstraction of the changes should be extracted to guess the intentions of the developer. [3] 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 became reusable on other files.

The currently 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* [4] has a refactoring engine that logs the changes performed by refactoring actions, if they were done via that engine, like renaming a variable or a class. That logs can be utilized further during the merge process.

Molhado is a refactoring-aware SCM system, that includes an *Eclipse* plugin *MolhadoRef* [5], 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, if so, they are removed from the change list. After that, *Molhado* can perform a textual merge by replaying the recorded refactoring together with other edit operations in order to propagate changes. Authors of *MolhadoRef* got better merge results with less human intervention compared to *CVS*.

Operation-based approaches can be very precise in recording the changes and replaying them, but sometimes the log files are unavailable. *RefactoringCrawler* [6] 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 [7] 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 also 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 traversing 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 operations, but 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 related 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 that is taken as input is semantically correct, it does not contain a lonely-changed reference. The advantage of our method over other object-oriented tools is that we support local variables.

Conclusions and Future Work

The presented approach takes closer to a semantically correct merge. But, there are a lot of other semantic related problems that were not addressed, however, huge number of compile-time errors can be reduced by the presented approach and it is going toward an automatic merge without human intervention. Our future work involves the reseach of the solutions to other semantic problems.

The approach can work in merging generated code with hand written code, where refactorings are not explicitly intended by the developers, but just caused by code generator, because some parameters were changed. As future work we also plan to improve the presented approach to create an efficient code generation tool with round-trip engineering support, that can be used in a designing environment, which applies bi-directional validated model to source transformations.

Acknowledgement

The fund of 'Mobile Innovation Centre' has partly supported the activities described in this paper.

References

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

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

[3] 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

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

[5] 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

[6] 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

[7] 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