Semistructured Merge: Rethinking Merge in Revision Control Systems

Sven Apel, Jörg Liebig,
Benjamin Brandl, Christian Lengauer
University of Passau, Germany

Christian Kästner
Philipps University Marburg, Germany

ABSTRACT

An ongoing problem in revision control systems is how to resolve conflicts in a merge of independently developed revisions. Unstructured revision control systems are purely text-based and solve conflicts based on textual similarity. Structured revision control systems are tailored to specific languages and use language-specific knowledge for conflict resolution. We propose semistructured revision control systems that inherit the strengths of both: the generality of unstructured systems and the expressiveness of structured systems. The idea is to provide structural information of the underlying software artifacts — declaratively, in the form of annotated grammars. This way, a wide variety of languages can be supported and the information provided can assist in the automatic resolution of two classes of conflicts: ordering conflicts and semantic conflicts. The former can be resolved independently of the language and the latter using specific conflict handlers. We have been developing a tool that supports semistructured merge and conducted an empirical study on 24 software projects developed in Java, C#, and Python comprising 180 merge scenarios. We found that semistructured merge reduces the number of conflicts in 60 % of the sample merge scenarios by, on average, 34 %, compared to unstructured merge. We found also that renaming is challenging in that it can increase the number of conflicts during semistructured merge, and that a combination of unstructured and semistructured merge is a pragmatic way to go.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Management, Measurement, Experimentation

Keywords

Revision Control, Version Control, Software Merging, Semistructured Merge, FSTMERGE, FEATUREHOUSE

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

1. INTRODUCTION

Revision control systems (a.k.a. version control systems) have a long tradition in software engineering and are a major means for managing revisions and variants of today's software systems [12, 14, 19]. A programmer creates a revision of a software system by deriving it from the base system or from another revision; a revision can be developed and evolve in isolation; and it can be merged again with the base system or another revision. A major problem of revision control is how to resolve merge conflicts that are caused by concurrent changes (e.g., when two developers independently change the same method) [17]. Zimmermann found in an empirical study of four large projects that 23 % to 46 % of all merge scenarios exhibit conflicts [22].

In recent years, two classes of revision control systems have emerged: (1) revision control systems that operate on plain text and (2) revision control systems that operate on more abstract and structured document representations. The first class is being used widely in practice because such systems are typically language-independent. Popular systems of this class are CVS, Subversion, and Git. Henceforth, we call them *unstructured revision control systems*. A problem of unstructured revision control systems is that, when conflicts occur, no knowledge of the structure of the underlying software artifacts is available. This makes it difficult to resolve certain kinds of conflicts, as we will illustrate.

The second class of revision control systems is explored mainly in academia with the goal of solving the conflict-resolution problems of unstructured revision control systems [11, 21]. The idea is to use the structure and semantics of the software artifacts being processed to resolve merge conflicts automatically [17]. These systems operate on abstract syntax trees or similar representations, rather than on plain program text, and incorporate all kinds of information on the underlying language. A drawback is that, relying on a particular language's syntax or semantics, they sacrifice language independence. Henceforth, we call them *structured revision control systems*.

There is a trade-off between generality and expressiveness of revision control systems [14]. A revision control system is general if it works with many different kinds of software artifacts. It is expressive if it is able to handle many merge conflicts automatically. Inspired by this trade-off, we propose a new class of revision control systems, called *semistructured revision control systems*, which inherit the strengths but not the weaknesses of unstructured and structured revision control systems. The idea is to increase the amount of information that a revision control system has at its disposal to resolve conflicts, while maintaining generality by supporting many languages. In particular, we concentrate on the merge process, so we speak of *semistructured merge*.

We developed a generic engine, called FSTMERGE, that merges different revisions of a software system based on the structure of

the software artifacts involved. Users can plug a new language into FSTMERGE by providing a declarative specification of their language's syntax (i.e., a grammar) enriched with information for conflict resolution. While this approach is not entirely language-independent, it is still quite general. First, it enables to resolve conflicts that are caused by differing orderings of program elements independently of the language. Second, it allows the user to plug in dedicated conflict handlers in order to define further (possibly language-dependent) resolution strategies. If, for whatever reason, there is no grammar available for a certain language, FSTMERGE can use a fallback solution to parse software artifacts line by line, which makes it equivalent to the unstructured approach.

Overall, we make the following contributions:

- Based on an analysis of the trade-off between generality and expressiveness, we propose the concept of semistructured merge, which combines the strengths of unstructured and structured merge. It is able to resolve ordering conflicts independently of the language, and it can be extended by conflict resolution handlers to define additional (language-dependent) conflict resolution strategies.
- We provide the tool FSTMERGE for semistructured merge as an extension of the FEATUREHOUSE tool infrastructure for software composition [4].
- By using FSTMERGE on artifacts written in Java, C#, and Python, we demonstrate its applicability and generality.
- We report on an empirical study on 24 software projects to quantify the benefits of semistructured merge and to identify open issues. In a nutshell, we found that semistructured merge can reduce the number of conflicts in 60 % of the sample merge scenarios by 34±21 %.[1] Furthermore, our study reveals that renaming is a challenge for semistructured merge that can increase the number of conflicts (in 26 % of the sample merge scenarios by 109±170 %), which we discuss.
- A combination of unstructured and semistructured merge reduces the number of conflicts in our study by 35 % compared to pure unstructured merge.

The latter three contributions are novel compared to an earlier workshop paper [6]. The key contribution is the empirical study whose goal has been to explore how far we can get with an approach that is centered around ordering conflicts and conflict handlers — the results are encouraging, as we will discuss. The prototypical tool FSTMERGE is under ongoing development; in particular, it was a means to conduct the study.

2. BACKGROUND AND RELATED WORK

There is a large body of work on revision control systems [12, 14, 19] and conflict resolution in software merging [17]. The purpose of a revision control system is to manage different revisions of a software system. Usually, revisions are derived from a base program or from other revisions. Users can check out revisions from the repository, make asynchronous changes, and check them in again. By branching the main development line (or trunk), a programmer can create independent revisions, which can be changed and evolve independently (e.g., to add and test new features). Independent revisions can be merged back with the base program or with other revisions, which may have been changed in the meantime.

The key issue we address in our work is *merge conflict resolution*. Conflicts may occur while merging independent changes. A major goal of research on this problem is to empower revision control systems to resolve merge conflicts automatically, that is, to reduce the number of conflicts that otherwise have to be resolved manually by the user [17]. First, we illustrate the problem of conflict resolution in unstructured merge. Then, we illustrate the ability of structured merge to resolve conflicts better than unstructured merge.

[1] We write $m \pm s$ as an abbreviation for the mean value m of a data set and its standard deviation s.

2.1 Unstructured Merge

To illustrate the conflict resolution problem, we use the running example of a simple stack implementation, as shown in Figure 1 (top). Henceforth, we call this program the base program or simply STACK. It contains a class Stack that implements interface Cloneable and that contains a field items and the two methods push and pop.

Now, suppose a programmer would like to add a new method top to Stack, but would like to develop this feature in its own branch, independently of the main branch (i.e., the base program). To this end, the programmer creates a branch with a new revision TOP. Furthermore, suppose another programmer adds subsequently a method size directly to the main branch by creating revision SIZE of the base program. Figure 1 (middle) presents code for the two revisions, each of which adds a new method to class Stack. Finally, suppose that, at some point in time, the two branches are merged again to combine both revisions including the new methods.

Merging the two branches involves merging the two revisions TOP and SIZE (solid arrows) on the basis of the common ancestor, the base program STACK (dashed arrow). This process is called a *three-way merge* because it involves three revisions [19]. In our example, the merge process reports a conflict that cannot be resolved automatically with unstructured merge. Figure 1 (bottom) contains the output of the merge tool of CVS for this example. The output shows that the tool is not able to merge the two new methods top and size, such that both can be present in the merged program, because it cannot determine the desired order of both methods. Similar conflicts are reported by the merge tools of Subversion and Git — the user has to merge the revisions manually.

This example is very simple but it suffices to illustrate the problems of unstructured merge. An unstructured merge tool operates solely on the basis of text lines or tokens. It identifies new text fragments with regard to the common ancestor (base program) and stores the common fragments before and after the new fragments. If the two revisions change or extend text in the same region, the system reports a conflict. That is, it is not able to decide how to merge the changes or extensions. In our example, the merge tool knows that two independent text fragments (which actually implement the two methods top and size) are added to the same location of the base program (which is enclosed by the two fragments that implement the methods push and pop). The problem is that the unstructured merge tool does not know that these fragments are Java methods and that a merge of the two is actually straightforward because their order within their enclosing class can be permuted safely. If the tool knew this, it would be able to resolve the conflict automatically. There are actually two solutions: include method top first and then method size, or vice versa.

2.2 Structured Merge

In the past, many tools have been proposed that leverage information on the artifact language to resolve as many conflicts as possible [17]. Westfechtel and Buffenbarger pioneered this field by proposing tools that incorporate structural information such as the context-free and context-sensitive syntax in the merge process [11, 21]. Researchers proposed a wide variety of structural comparison and merge tools including tools specific to Java [8] and C++ [15]. Some tools even consult additionally semantic information of the language [10].

Let us illustrate the abilities of structured merge by a further ex-

Base Program STACK

```
1  import java.util.LinkedList;
2  public class Stack<T> implements Cloneable {
3    private LinkedList<T> items = new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7    public T pop() {
8      if(items.size() > 0) return items.removeFirst();
9      else return null;
10   }
11 }
```

branch

Revision SIZE

```
1  import java.util.LinkedList;
2  public class Stack<T>
         implements Cloneable {
3    private LinkedList<T> items =
           new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7    public int size() {
8      return items.size();
9    }
10   public T pop() {
11     if(items.size() > 0) return
             items.removeFirst();
12     else return null;
13   }
14 }
```

Revision TOP

```
1  import java.util.LinkedList;
2  public class Stack<T>
         implements Cloneable {
3    private LinkedList<T> items =
           new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7    public T top() {
8      return items.getFirst();
9    }
10   public T pop() {
11     if(items.size() > 0) return
             items.removeFirst();
12     else return null;
13   }
14 }
```

merge

$merge_{unstructured}$(TOP, STACK, SIZE)

```
1  import java.util.LinkedList;
2  public class Stack<T> implements Cloneable {
3    private LinkedList<T> items = new LinkedList<T>();
4    public void push(T item) {
5      items.addFirst(item);
6    }
7  <<<<<<< Top/Stack.java
8    public T top() {
9      return items.getFirst();
10   }
11 =======
12   public int size() {
13     return items.size();
14   }
15 >>>>>>> Size/Stack.java
16   public T pop() {
17     if(items.size() > 0) return items.removeFirst();
18     else return null;
19   }
20 }
```

Figure 1: Merging the revisions SIZE and TOP with unstructured merge

Base Program STACK

```
1  import java.util.LinkedList;
2  public class Stack<T> implements Cloneable { ... }
```

branch

Revision FLUSHABLE

```
1  import java.util.LinkedList;
2  import java.io.Flushable;
3  public class Stack<T>
       implements Cloneable,
       Flushable {
4    ...
5    public void flush() { ... }
6  }
```

Revision SERIALIZABLE

```
1  import java.util.LinkedList;
2  import java.io.Serializable;
3  public class Stack<T>
       implements Cloneable,
       Serializable {
4    private static final long
         serialVersionUID = 42;
5    ...
6  }
```

merge

$merge_{unstructured}$(SERIALIZABLE, STACK, FLUSHABLE)

```
1  import java.util.LinkedList;
2  <<<<<<< Serializable/Stack.java
3  import java.io.Serializable;
4  =======
5  import java.io.Flushable;
6  >>>>>>> Flushable/Stack.java
7  <<<<<<< Serializable/Stack.java
8  public class Stack<T> implements Cloneable, Serializable {
9    private static final long serialVersionUID = 42;
10 =======
11 public class Stack<T> implements Cloneable, Flushable {
12 >>>>>>> Flushable/Stack.java
13   private LinkedList<T> items = new LinkedList<T>();
14   public void push(T item) {
15     items.addFirst(item);
16   }
17   public T pop() {
18     if(items.size() > 0) return items.removeFirst();
19     else return null;
20   }
21   public void flush() { ... }
22 }
```

Figure 2: Merging the independently developed revisions FLUSHABLE and SERIALIZABLE with unstructured merge

ample in Figure 2. Suppose we have the base stack implementation and we create two independent revisions, one that enables stack objects to be serialized (revision SERIALIZABLE) and another that allows programmers to flush the elements of the stack to a data stream (revision FLUSHABLE). Figure 2 depicts excerpts of the revisions. Merging the two revisions with the base program using unstructured merge causes two conflicts. First, the system is not able to merge the two new import statements and, second, it is not able to merge the implements clauses of the two revisions. Figure 2 (bottom) shows the conflicts as reported by the merge tool of CVS.

In contrast, a structured revision control system that knows that the base program and the revisions are written in Java is able to resolve such conflicts automatically. It knows that the order of imports does not matter and that, in this case, implements clauses can be combined by union.

Beside the types of conflicts we have seen so far, there are other types of conflicts that can be resolved by structured revision control systems on the basis of language-specific knowledge. For example, a for loop in Java consists of a head and a body, and the head consists of three parts. This information is useful when two revisions modify disjoint parts of the head. Even if a structured merge cannot merge a conflict automatically, it may have information that assists the programmer in resolving the conflict (e.g., even if the heads of two for loops cannot be merged, a structured merge tool knows that the conflict occurs in a loop head and not in an arbitrary line of text).

2.3 Generality vs. Expressiveness

The previous discussion reveals that there is a trade-off between generality and expressiveness of revision control systems, as has been observed before [14]. Unstructured revision control systems are very general. They can be used with every kind of (textual) software artifact, but they are not able to resolve conflicts that require

knowledge on the language of the artifacts involved. In contrast, a structured revision control system is typically tailored to a particular language. So, it would be possible to build a revision control system for Java that can resolve the conflicts we have discussed so far and, in addition, many other conflicts. However, such a system would be less useful in a setting in which a software system consists of artifacts written in many different languages (e.g., Java, JSP, Python, SQL, and Ant in a Web application).

This trade-off motivates us to explore the space between unstructured and structured revision control systems. Can we design a system that is able to handle a wide variety of software artifacts and that has enough information on these artifacts to resolve a reasonable number of conflicts automatically? A trivial solution would be to develop one structured revision control system for every artifact type that occurs in a software project, respectively. A problem with this naive approach is that it is very tedious and error-prone. Moreover, in many cases, not all artifact types can be anticipated.

2.4 Finding a Balance

Previous attempts to find a proper balance between structured and unstructured merge have some limitations. Niu et al. have proposed a merge approach that is both structured and independent of the language [18]. They represent programs as graphs and use graph morphisms to capture the relationships between the structural elements of the programs to be merged. The focus on graph structures, instead of on tree structures, is the biggest difference to our approach. The question of the scalability of graph-based algorithms may be raised, as general graph algorithms are usually more complex than tree algorithms. Niu et al. tested their approach only using a single C program of moderate size. It is not clear how their tool performs on larger programs and how difficult it is to provide support for new languages.

The approach of Dig et al. assumes a setting in which refactorings of the documents under revision control are recorded and replayed on demand [13]. That is, their approach is not applicable to scenarios in which this additional information is not available, which appears to be the rule rather than the exception in practice.

Other approaches require that the documents to be merged come with a formal semantics [9, 16], which is not always feasible in practice because, even for mainstream languages such as C++, there is no formal semantics available. We explore whether we can reduce the number of conflicts without these limitations.

3. SEMISTRUCTURED MERGE

3.1 Overview

Program Structure Trees. The basic idea of semistructured revision control systems is to represent software artifacts as trees and to provide information on how the nodes of a certain type (e.g., methods or classes) and their subtrees are merged. We call such a tree, which is essentially a parse tree, a *program structure tree* (or *feature structure tree* [4]). In Figure 3 (a), we show a simplified program structure tree of the base program STACK and, in Figures 3 (b) and 3 (c), simplified program structure trees of revisions TOP and SIZE. It is important to note that not all structural information is represented in the tree. For example, there are no nodes that represent statements or expressions. But this information is not lost; it is contained as plain text in the leaves (not shown). So a program structure tree is not necessarily a full parse tree; it abstracts from some details and contains them as text. In our setting, the order of child nodes of a common parent is arbitrary. That is, it may change without affecting the program semantics. This is a key

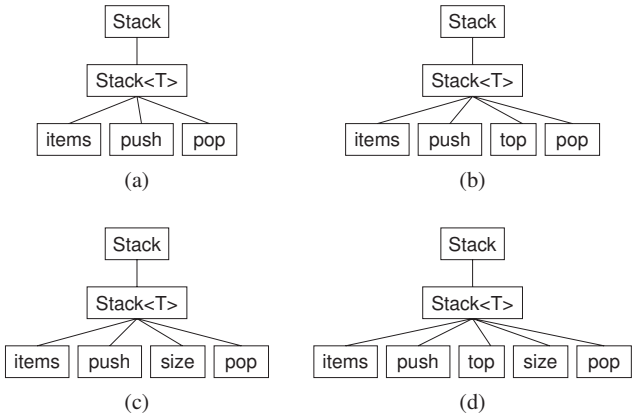

Figure 3: Revisions of the stack example as program structure trees

property when resolving conflicts that rely on the element order, as we will explain shortly.

The choice of which kind of structural element is represented by a distinct node depends on the expressiveness which we want to attain with semistructured merge. Let us explain this choice by means of the stack example. Taking the three program structure trees as input, a merge tool can produce the desired output without conflict only by superimposing the trees, as shown in Figure 3 (d).[2] Why does this algorithm work? It works because the order of method declarations in Java does not matter, so superimposing the trees just adds the two new methods top and size in any possible order. If the two revisions added methods with identical signatures, the tool would have to merge the statements of their bodies. This would be more difficult since their order matters (and statements do not have unique names) [1, 5]. Even with all the knowledge on the Java language at our disposal, there remain cases in which we cannot say how to merge sequences of statements. This is the reason why we choose to represent methods as opaque leaves and their statements as text content; in other languages, we may choose a different granularity.

Conflict Types. The ability of semistructured merge to resolve certain conflicts is based on the observation that the order of certain elements (e.g., of classes, interfaces, methods, fields, imports, and so on) does not matter. We call such conflicts *ordering conflicts*. A merge algorithm that resolves just ordering conflicts automatically is defined more simply than a fully structured merge. A semistructured merge uses an *abstraction* of the structure of the document, which provides just enough information to identify ordered items.

Furthermore, in the example of Figure 2, we see that unstructured merge is not able to combine the differing implements clauses of two revisions of a class. With semistructured (and structured) merge, we are able to achieve this because we know that lists of types can be unified *in this case*. The routine that resolves conflicts of a special kind is called a *conflict handler*. To distinguish conflicts for which we use special conflict handlers from the conflicts that occur due to ordering issues, we call the former *semantic conflicts* and the latter, as before, ordering conflicts. We do not claim that it is possible to define a conflict handler for every case. However, at least, a conflict handler can assist the developer in identifying and resolving conflicts.

But what do we do with program elements which we do not

[2]Superimposition merges trees recursively, beginning from the root, based on structural and nominal similarities [4].

```
1  @FSTNonTerminal(name="{Type}")
2  ClassDecl : "class" Type ImplList "{"
3    (FieldDecl)* (ClassConstr)* (MethodDecl)*
4  "}";
5  @FSTTerminal(name="Impl", merge="ImplUnion")
6  ImplList : "implements" Type ("," Type)*
7  @FSTTerminal(name="{ID}({Params})", merge="LineBasedMerge")
8  MethodDeclaration : Type ID "(" (Params)? ")" "{"
9    (Statement)*
10 "}"; ...
```

Figure 4: An excerpt of a simplified Java grammar with annotations

```
1  public class ImplUnion {
2    public final static String HANDLER = "ImplUnion";
3    public static void compose(FSTTerminal a, FSTTerminal b,
           FSTTerminal comp, FSTNonTerminal parent) {
4      String ifcA = a.getBody();
5      String ifcB = b.getBody();
6      String ifcComp = ifcB + ifcA;
7      ...
8    }
9  }
```

Figure 5: An excerpt of a conflict handler for merging implements lists in Java

know how to merge, such as method bodies with statements? The answer is simple: we represent the elements as plain text and use conventional unstructured merge. That is, if a conflict occurs inside a method body, we do not attempt to resolve it automatically — much like in unstructured merge.

Note that superimposition alone is not sufficient to implement semistructured merge because revisions may remove program elements. For example, one or both revisions of STACK may remove method push. The result should be a stack without this method. To this end, the merge algorithm has to check whether one or both revisions remove elements, and so on. In our tool, we implemented a set of standard rules for removing elements (following RCS's merge), whose description we omit here because they do not add anything to the discussion.

3.2 System Architecture and Annotations

Our system consists of two parts: (1) a generic engine that knows how to identify and resolve certain conflicts and (2) a small abstract specification — for each artifact language — of the program's or document's elements whose order does not matter or for which special conflict handlers are available. The abstract specification of a document structure is given by an annotated grammar of the language. Most of the work is done by the generic merge engine, using the grammar as a guide. This architecture makes it relatively easy to include new languages by providing proper abstract specifications, compared to implementing a complete structured merge tool from scratch.

To illustrate the role of annotations, consider the excerpt of a simplified Java grammar in Figure 4. It contains a set of production rules. For example, rule ClassDecl defines the structure of classes containing fields (FieldDecl), constructors (ClassConstr), and methods (MethodDecl). Production rules may be annotated with @FSTNonTerminal and @FSTTerminal. The former annotation defines that elements corresponding to the rule are represented as nodes in the corresponding program structure tree, there may be subnodes, and the order of elements or nodes is arbitrary. In our example, we annotate the rule for class declarations with @FSTNonTerminal because classes may contain further classes, methods, and so on, and the order of classes in a file or package may vary. The annotation parameter name is used to assign a name to the corresponding node in the program structure tree. Annotation @FSTTerminal is like annotation @FSTNonTerminal except that subelements are represented as plain text and that the developer can decide how code corresponding to the element in question is merged. For example, we annotate the rule for method declarations with @FSTTerminal to represent their inner statements by plain text, as explained before. Annotation parameter merge defines that the method's content is merged by a classic line-based merge algorithm, as used in unstructured merge.

A further interesting example is the rule for describing the syntax of implements lists (Fig. 4, Lines 5–6). This rule is annotated with @FSTTerminal, so the subelements (i.e., list of type names) are represented as plain text. But, in contrast to method declarations, we state that implements lists are merged by a special conflict handler that combines them by union (merge="ImplUnion"). This conflict handler is specific to Java's implements list and provided as a plugin for FSTMERGE (via FEATUREHOUSE's plugin mechanism). In Figure 5, we show an excerpt of a possible implementation in our tool. Note that the name of the handler, defined in constant HANDLER (Line 2), matches the name of the selected handler in the annotated grammar (Fig. 4, Line 5).

The example of Figure 5 illustrates the simplicity of defining conflict handlers. It is important to note that conflict handlers are not always language-independent but optional (if not provided, unstructured merge is used). Conflict handlers can be used not only to resolve conflicts, but also to provide structured and language-specific information on the kind of conflict and potential conflict resolution strategies, in the case a conflict cannot be resolved automatically using the information available. For example, we could use a conflict handler to specify how the parts of a for loop head are merged and, if this is not possible, to provide information on the kind and cause of the conflict.

3.3 Balancing Generality and Expressiveness

Semistructured merge is a combination of unstructured merge and structured merge. Elements whose order is arbitrary are represented by nodes and can be merged by superimposition. Elements whose order is not arbitrary are represented by text and merged either by unstructured merge or by special conflict handlers. That is, we have a choice per language of which elements are to be represented by nodes and which by text and which elements are treated by conflict handlers. With a coarse granularity, we have simple program structure trees (e.g., with Java classes as leaves) but cannot resolve conflicts at the method level. That is, the expressiveness of the merge algorithm is limited. A finer granularity leads to more complex trees (e.g., with Java fields and methods as leaves) but also allows a merge tool to resolve ordering conflicts automatically (or via conflict handlers). The result is a higher expressiveness of the merge algorithm.

As we have illustrated, an annotated grammar contains sufficient information to guide a revision control system in merging Java artifacts. But how does this approach facilitate generality? Indeed, for a language to be supported, we need some information in the form of an annotated grammar and, if desired, special conflict handlers, so the tool is not entirely language-independent. But, for many languages, such a grammar is easily provided, since standard grammars in Backus-Naur-Form are available on the Web for many languages, and adding annotations is a matter of hours, at most. Also, implementing special conflict handlers in the form of Java code is a matter of hours (e.g., for concatenating implements lists, a few lines of Java code are sufficient).

3.4 Summary

To summarize, semistructured merge is more expressive than unstructured merge, since certain conflicts can be resolved automatically based on information on the underlying languages. And semistructured merge is more general than structured merge, since a wide variety of languages can be supported on the basis of an annotated grammar, which needs to be provided only once per language. If, for whatever reason, no information on a given language is available, semistructured merge behaves exactly like unstructured merge, parsing the corresponding software artifact line by line.

4. IMPLEMENTATION

To demonstrate the feasibility of our approach, we implemented a semistructured-merge tool and used it to merge artifacts written in Java, C#, and Python.

4.1 Generic Merge Engine

We have implemented a first prototype of a semistructured-merge tool, called FSTMERGE, which is able to resolve ordering conflicts and which can be extended with special conflict handlers. FSTMERGE and the sample programs of our empirical study are publicly available on the project's Web site.[3]

FSTMERGE takes advantage of the existing tool infrastructure FEATUREHOUSE [4], as illustrated in Figure 6. The tool FSTGENERATOR generates almost all code that is necessary for the integration of a new language into FSTMERGE. FSTGENERATOR expects the grammar of the language in a specific format, called FEATUREBNF, of which we have already shown an example in Figure 4. We have extended FSTGENERATOR and FEATUREBNF to support annotations relevant to semistructured merge.

Using a grammar written in FEATUREBNF, FSTGENERATOR generates an LL(k) parser (that produces program structure trees) and a corresponding pretty printer, both of which are then integrated into FSTMERGE. Furthermore, developers have the opportunity to plug in special conflict handlers and to activate them in the grammar (using annotation parameter merge). After the generation step, FSTMERGE proceeds as follows: the generated parser takes the base program and two revisions written in the target language and produces a program structure tree for each program; FSTMERGE performs the semistructured merge as explained before (the trees are superimposed, deletions are handled separately, and special conflict handlers or a conventional unstructured merge are applied to the leaves); the generated pretty printer writes the merged revisions to disk.

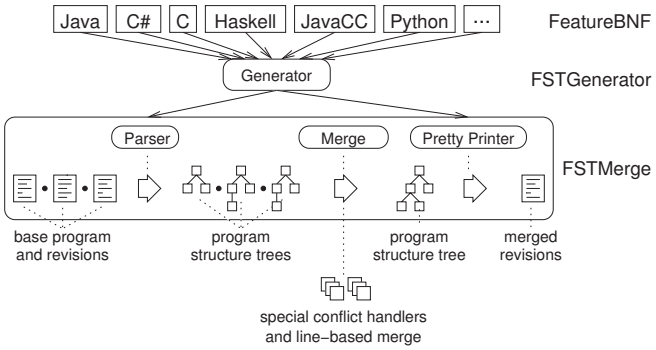

Figure 6: The architecture of FEATUREHOUSE

[3] http://www.fosd.de/SSMerge/

4.2 Language Plugins

We have tested FSTMERGE with three languages: Java, C#, and Python. All three languages are common in projects that use revision control systems and all are sufficiently complex and different from each other to demonstrate the practicality and generality of our approach. As a basis, we used publicly available grammars that we adapted and annotated for our purposes. Beside technical problems with the Python grammar (the Python grammar relies on indentation), plugging the three languages into FEATUREHOUSE and FSTMERGE was easy. Annotating and testing them was a matter of a few hours. Furthermore, we implemented special conflict handlers for merging implements, extends, and throws lists and for modifiers, mainly for displaying information on the conflicts. Typically, the implementation of a conflict handler requires only a few lines of well-separated Java code and follows the pattern of the handler shown in Figure 5.

5. EMPIRICAL STUDY

To quantify the benefits of our approach and to discover open issues, we conducted an empirical study on merging and conflict resolution in a number of software projects.

5.1 Motivation & Methodology

Due to our experience with FEATUREHOUSE in software product-line engineering [2–4, 7], we expect that integrating further languages is easy. The interesting issue is whether, in its current form, semistructured merge can play to its strengths in real software projects. The point is that concentrating on ordering and certain semantic conflicts is a fairly simple approach. By means of examples, we have demonstrated that semistructured merge is able to resolve conflicts that cannot be resolved with unstructured merge. But how frequently do such conflicts occur in real software projects? How far can we get with such a simple approach?

Hypothesis: *Many of the conflicts that occur in merging revisions are ordering conflicts, which can be resolved automatically with semistructured merge. An additional fraction of conflicts can be resolved with conflict handlers.*

Although there is some evidence that revisions often involve additions of larger structures such as entire functions [20], we need a substantial set of data to answer the question systematically. To this end, we used both semistructured merge and unstructured merge in a number of merge scenarios and compared the resulting numbers of conflicts, lines of conflicting code, and conflicting files. Unfortunately, we could not compare semistructured merge with structured merge because there are simply no practical tools available, least of all for Java, C#, and Python.[4] Also we could not compare our approach with tools that build on formal semantics [9, 16] or on refactoring replaying [13] because our approach aims at a broader scenario in which this information is not available.

Beside the quantitative analysis, we looked at selected samples of merged code in order to learn about the influence of the merge approach on the resulting code structure. We found a number of differences that may have an impact on code comprehension, which we discuss in Section 5.4.

[4] We are aware only of the *academic* prototypes for C of Westfechtel [21] and Buffenbarger [11]. These tools are not able to resolve ordering conflicts (personal communication with Westfechtel).

5.2 Sample Projects

Criteria. An important issue is how to select merge and conflict scenarios in a comparison of semistructured and unstructured merge. Developing our own case study would leave too much room for bias. Using only one or two scenarios would threaten external validity (i.e., we would not be able to generalize our findings). Thus, we decided to base our study on a substantial number of open-source projects developed by practitioners for real applications. We explored SourceForge, a popular open-source software portal, for candidates. We had two criteria: (1) the projects must be of reasonable but varying sizes and (2) either semistructured merge or unstructured merge must produce at least one conflict. To this end, we analyzed the revision history of the candidates including logs for real and realistic merges, which we call henceforth *merge scenarios*. Technically, we used Subversion to browse the revision histories and to check out revisions. In principle, we could have used any other state-of-the-art revision control system, but we were most familiar with Subversion.

Identifying Merge Scenarios. We have been considering two kinds of merge scenarios: (1) merges that have actually been performed in the revision history of a project and (2) merges that are realistic considering the history of a project.

Based on the logs, we were able to extract information on merges that actually have been performed by the developers and to obtain the revisions involved. Since Subversion has no standard log entry type for merges,[5] we used comments of the developers that point clearly to merges.

Beside merges that actually happened, we searched for merges that could have been performed (the logs are not clear about this) or that are realistic considering the revision history — to increase the sample size. One kind of indicator is a sequence of multiple, alternating changes in different branches (e.g., trunk → branch → trunk). This pattern indicates concurrent development and points to potential conflict scenarios (as long as the changes in different branches are not identical). A second indication is that developers evolve only a single branch in many revisions. Merging the branch back to the trunk is a classic scenario in revision control (e.g., to merge new features that have been added and tested in a separate branch with the trunk) and is likely to produce conflicts when the trunk has changed in between (typically, the more changes, the more conflicts).

Technically, a merge scenario is a triple consisting of a base revision, a left branch, and a right branch. The base revision marks the point in time in which the left or right branch was copied from the other. This way, we ensure that both branches have a common ancestor, which is necessary for a three-way merge. Furthermore, we check the two independently evolved branches out at the same point in the revision history, which is a prerequisite for merging them in practice.

In Table 1, we list information on the sample projects including name, domain, number of lines of code, number of merge scenarios, and programming language. Since Java is widely used, we were able to locate ten projects with ten conflict scenarios each. For C# and Python, we located fewer candidates with fewer conflict scenarios. Overall, our sample consists of 24 projects with, in summary, 180 merge scenarios comprising, summed over all scenarios, 50 million lines of code. All merge scenarios are available and documented on the project's Web site.

[5]This entry type is available since version 1.5, but all sample projects use earlier versions or did not take full advantage of this feature.

Project	Domain	LOC	MS	Lang.
AutoWikiBrowser	semi-autom. Wikipedia editor	63 K	9	C#
BitPim	mobile phone synchronization	180 K	7	Python
CruiseControl.NET	continuous integration server	148 K	9	C#
DrJava	development environment	89 K	10	Java
emesene	instant messaging client	29 K	1	Python
Eraser	secure data removal tool	18 K	6	C#
eXe	eLearning XHTML editor	98 K	9	Python
FreeCol	turn-based strategy game	86 K	10	Java
FireIRC	IRC client	17 K	1	C#
GenealogyJ	editor for genealogic data	56 K	10	Java
iFolder	directory synchronization	68 K	4	C#
iText	PDF library	71 K	10	Java
JabRef	BibTeX entry manager	75 K	10	Java
jEdit	programmer's text editor	107 K	10	Java
JFreeChart	chart library	149 K	10	Java
Jmol	viewer for chemical structures	135 K	10	Java
matplotlib	plotting library	86 K	10	Python
NASA WorldWind	virtual globe	193 K	8	C#
PMD	bug finder	71 K	10	Java
Process Hacker	process viewer	99 K	1	C#
RSS Bandit	news feeds reader	116 K	4	C#
SpamBayes	Bayesian anti-spam classifier	47 K	6	Python
SQuirrelSQL	graphical SQL client	218 K	10	Java
Wicd	network manager for Linux	5 K	5	Python

Table 1: Overview of the sample projects (all available on `http://sourceforge.net/`); LOC: lines of code; MS: number of merge scenarios.

5.3 Results

In Table 2, we depict the results of our analysis in terms of the numbers of conflicts, conflicting lines of code, conflicting files, and semantic conflicts. For the purpose of the study, we defined conflict handlers for 54 structural elements of Java, C#, and Python (implements lists, modifiers, and so on). Typically, the handlers are very simple and only flag a semantic conflict. So, we did not implement specific resolution strategies but we just counted situations in which they can be applied, which is sufficient for the quantitative analysis. Details on the conflict handlers are available on the project's Web site.

Due to the sheer amount of data, we provide for each project mean values over all merge scenarios. Detailed information on each merge scenario (including the number of conflicts, conflicting lines of code, conflicting files, and semantic conflicts per file and per merge scenario in the form of tables and diagrams) are available on the project's Web site.

Based on the results, we classify the projects into two categories, according to which numbers are reduced by semistructured merge compared to unstructured merge:

1. The number of conflicts, conflicting lines of code, and conflicting files are reduced or on the same order of magnitude.
2. The number of conflicts or conflicting lines is significantly increased (highlighted in Tab. 2), but other numbers are reduced.

We have highlighted projects that fall into the second category with a gray background in Table 2.

Compared to unstructured merge, semistructured merge (including conflict handlers) can reduce the number of conflicts in 60 % of the sample merge scenarios by 34±21 % (the number of conflicting lines of code is reduced in 82 % of the sample merge scenarios by 61±22 %; the number of conflicting files is reduced in 72 % of the sample merge scenarios by 28±12 %).

Remarkably, in 26 % (46 of 180) of the sample merge scenar-

Project	conflicts		confl. LOC		confl. files		sem. confl.	
	UM	SM	UM	SM	UM	SM	UM	SM
AutoWikiBrowser	20	17	418	362	6	6	–	0
BitPim	29	25	5240	471	6	3	–	4
CruiseControl.NET	26	16	1133	279	14	8	–	5
DrJava	20	16	293	210	7	7	–	0
emesene	7	4	400	17	7	4	–	17
Eraser	24	24	2146	637	12	11	–	0
eXe	5	3	112	35	4	2	–	2
FireIRC IRC Client	5	3	36	23	3	2	–	0
FreeCol	237	213	4606	3265	63	58	–	1
GenealogyJ	19	9	489	146	9	6	–	0
iFolder	44	10	228	156	6	4	–	34
iText	219	706	113757	35109	216	179	–	262
JabRef	75	49	1701	1782	28	24	–	2
jEdit	7	6	105	80	3	3	–	0
JFreeChart	560	361	18142	5500	193	109	–	21
Jmol	94	112	20094	6869	40	33	–	4
matplotlib	57	15	500	466	13	9	–	16
NASA WorldWind	23	36	5946	2328	12	9	–	10
PMD	55	371	18469	6596	48	36	–	17
Process Hacker	1	1	4	4	1	1	–	1
RSS Bandit	65	51	4086	1478	12	8	–	32
SpamBayes	25	22	1166	343	10	7	–	2
SQuirrelSQL	26	24	2620	579	16	13	–	1
Wicd	10	9	1267	218	4	3	–	2

Table 2: Mean conflicts per scenario: numbers of conflicts, conflicting lines of code, conflicting files, and semantic conflicts for semistructured merge (SM) and unstructured merge (UM).

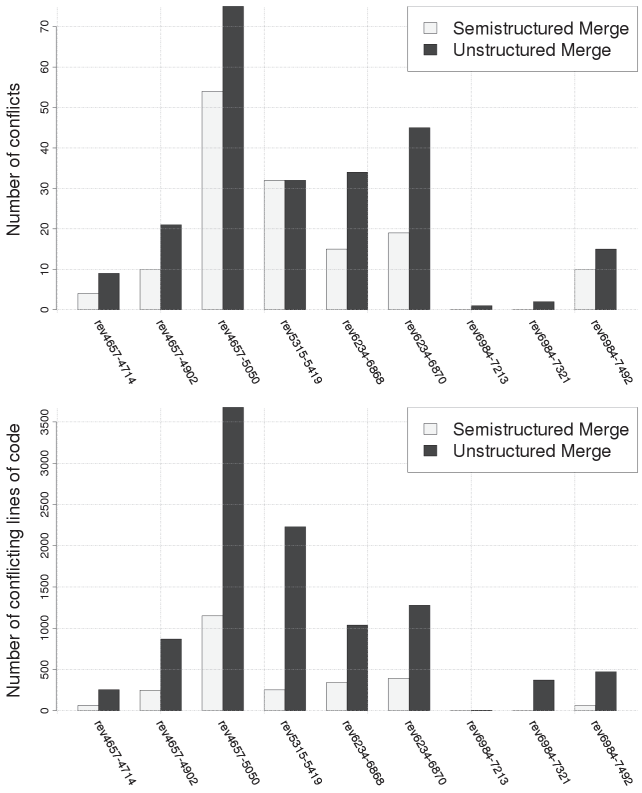

Figure 7: Results for CruiseControl.NET (first category)

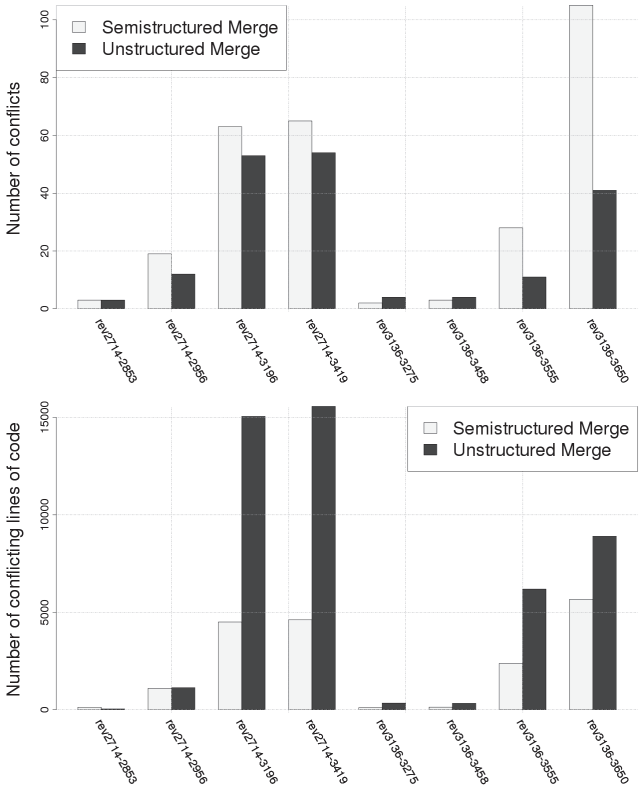

Figure 8: Results for NASA WorldWind (second category)

ios, semistructured merge increases the number of conflicts by 109±170 %. We found this result counter-intuitive at first: Why should additional information incur additional conflicts? In Section 5.4, we explain that renaming is the reason for the increased number of conflicts (and conflicting lines of code) in the projects of the second category.

To present more than just the mean values and to explore the reasons for the differences of the individual projects, we had a closer look at some representatives. In Figure 7, we display the numbers of conflicts and conflicting lines of code of all merge scenarios of CruiseControl.NET (first category).[6] The numbers show that semistructured merge can play to its strengths in CruiseControl.NET; the same applies to 18 other projects of our sample (all projects without gray cells in Table 2).

In Figure 8, we display the numbers of NASA WorldWind (second category). Interestingly, semistructured merge produces more conflicts than unstructured merge, which we did not expect initially. However, the number of conflicting lines of code is reduced significantly. This is similar for all projects in which the number of conflicts is increased. In Section 5.4, we discuss reasons for the diverging numbers.

Finally, we were surprised that the numbers of semantic conflicts are rather low compared to the numbers of ordering conflicts, especially when considering the quite high number of 54 elements that we handle with special conflict handlers. A notable difference is iText, which we discuss in Section 5.4.

[6]The name of a merge scenario is composed of the revision number of the base revision and the revision number of the left and right branch (which are equal, as explained in Sec. 5.2). For example, rev0815-4711 denotes a merge of the left and right branch of revision 4711 based on the base revision 0851.

5.4 Discussion

We found many scenarios in which semistructured merge is superior to unstructured merge in that it reduces the number of conflicts, conflicting lines of code, and conflicting files. So, we can confirm our hypothesis: many of the conflicts that occur in merging revisions are ordering conflicts, which can be resolved automatically with semistructured merge. We also found that an additional fraction of conflicts can be potentially resolved with language-specific conflict handlers.

Nevertheless, the fact that, in some cases, semistructured merge increases the number of conflicts or conflicting lines of code surprised us. We had expected semistructured merge to be at least as good as unstructured merge. To address this issue and to learn more about the nature of semistructured merge, we reviewed the merged revisions of all projects manually. Next, we provide a summary of our findings.

Renaming. An analysis of the projects of the second category revealed that renaming poses a challenge for semistructured merge. The point is that semistructured merge uses superimposition to merge revisions. If a program element is renamed in one revision (in one of the branches), the merge algorithm is not aware of this fact and cannot map the renamed element to its previous version. This results in a situation in which we have in one branch an empty or non-existent element. In Figure 9, we illustrate a situation in which semistructured merge produces more conflicts and conflicting lines than unstructured merge. In the base revision, we have a simple stack. In revision SOMECHANGE, method push of Stack has been changed. In revision RENAMING, class Stack has been renamed to RenamedStack. Unstructured merge does not produce any conflicts because the changes to the method and the class name are located in different lines. Semistructured merge identifies merge partners by name, so it treats class Stack and class RenamedStack as distinct elements. The result of semistructured merge is shown at the bottom of in Figure 9. It contains a conflict comprising four lines of code.

The worst case is that a directory is renamed — instead of a class, as in the stack example. This happened, for instance, in PMD and Jmol. In these cases, unstructured merge reports, for each file of the directory, a large conflict because it cannot map the files of the renamed directory to the corresponding files of the other branch and uses empty files instead. The same happens in semistructured merge, except that the conflicts are not reported per file but per method or constructor in the file. This results in more conflicts but the overall number of conflicting lines is smaller than in unstructured merge. The reason is that unstructured merge flags entire files as conflicts and semistructured merge only individual structural elements such as methods. We discuss how to handle renaming better in Section 5.6.

Semantic Conflicts. We did not find many semantic conflicts, compared to other conflicts. An exception is iText. Examining iText's source code, we found that, at some point, the developers switched from Java 1.4 to Java 1.5. This transition involved renaming a large number of raw types to generic types (e.g., from LinkedList to LinkedList<String>). Thus, the cause for the exceptionally high number of semantics conflicts is again a form of renaming.

Structural Boundaries. An observation we made when examining the results of unstructured and semistructured merge is that semistructured merge, due to its structure-driven and fine-grained nature, leads always to conflicts that respect boundaries of classes, methods, and other structural elements. This is not the case for unstructured merge. We found that respecting structural boundaries

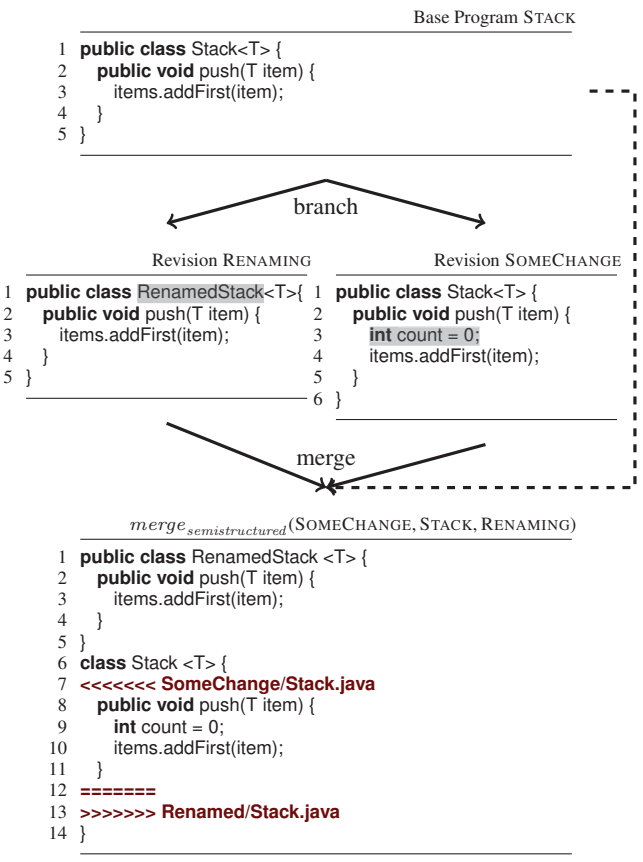

Figure 9: Semistructured merge in the presence of renaming

(i.e., aligning the merge with the program structure) is beneficial, because, this way, we could understand conflicts in terms of the underlying structure, even though this may result in more conflicts (e.g., one per method instead of one per file).

Automation. A goal of research on software merging is to minimize the number of conflicts that require manual intervention to be resolved [17]. However, an automatic resolution of conflicts has to be taken with a grain of salt. For example, if two developers add independently two methods that are intended to be the same, but the conflict resolution recognizes them as different, this mismatch goes unnoticed. This can happen in unstructured, structured, and semistructured merge. It is desirable to minimize the amount of manual intervention, but it is useful to keep information on conflict resolutions. Semistructured merge has benefits in this regard as it knows about the structural elements involved, not only about text.

5.5 Threats to Validity

Construct Validity. Although the output of semistructured merge in the presence of renaming is not satisfactory (cf. Fig. 9), it still allows us to detect conflicts properly and to incorporate them in our data. A threat to construct validity is that the number of conflicting lines of code may be estimated too low, because the renamed element is not considered (cf. Fig. 9). However, after code inspection we found that this fact is negligible for our conclusions.

Internal Validity. A threat to internal validity is that the results we obtained are influenced by (hidden) variables other than the kind

of merge (which is the independent variable). Due to the simplicity of our setting, we can largely rule out such confounding variables. We applied semistructured and unstructured merge to the same set of merge scenarios and counted the number of conflicts subsequently on the merged code. Furthermore, we used a comparatively large sample (24 projects, 180 merge scenarios, 50 million lines of code) to rule out confounding variables such as programming experience, programming style, and the difference between domains.

A potential threat to internal (and external) validity is our approach of selecting conflict scenarios. The problem is that none of the projects we found contains explicit information on merges in its logs. So, we had to search the logs for comments that indicate merge scenarios (such comments are fortunately quite common). To increase the number of conflict scenarios, we considered further merge scenarios based on a number of patterns in the revision hierarchy (e.g., small alternating changes in two independent branches or branches that end abruptly) — see Section 5.2. Overall, we are confident that the conflict scenarios we selected reflect a broad spectrum of the current practice in revision control (due to the large sample) and thus our results are reliable in this respect.

One can speculate about the influence of the merge approach on patterns of concurrent development. Developers may avoid certain programming and change patterns in order not to run into conflicts when using unstructured merge. The availability of semistructured merge may influence developers in the choice of their patterns because more conflicts can be resolved.

Even though a comprehensive analysis of code quality is well beyond the scope of the paper, a look at samples of merged code made us confident that the resulting code structure of semistructured merge can aid code comprehension as the merge process follows the structural boundaries of the artifacts involved. However, a series of controlled experiments is necessary to provide a definite answer to this question.

External Validity. A common issue is to what extent the external validity of our study relies on the selection of samples. Can we generalize to other projects in other domain and written in other languages? To increase external validity, we collected a substantial number of projects and merge scenarios written in different languages and of different domains. This does not guarantee that we will arrive at the same picture in other projects, domains, and languages, but we are confident that we have covered a broad spectrum of merge scenarios.

5.6 Open Issues & Perspectives

We see some interesting open issues. A first issue is the role of refactoring. So far, we have not addressed changes, such as the renaming of methods or classes, systematically. We found that renaming can increase the number of conflicts substantially, compared to unstructured merge. The reason is that superimposition relies on nominal equality. We see two ways of dealing with this problem. First, we could use unstructured merge instead of semistructured merge for files that contain renamings. A merge tool can simply compare the numbers of conflicts produced by semistructured merge and by unstructured merge on a per-file basis and choose the better alternative. Since merging is done in linear time, the overhead should be acceptable. For our sample, the combination of unstructured and semistructured merge reduces the number of conflicts, in sum, by 35 % compared to pure unstructured merge (and by 50 % compared to pure semistructured merge). A merge in our setting takes about 15 min but, compared to the time for resolving conflicts manually, the overhead for applying both kinds of merge can be safely neglected.

Additionally, like in the approach of Dig et al. [13], we could trace renamings and pass this information to the revision control subsystem. This way, pairs of corresponding elements can be matched, even though they have different names, and the conflict of Figure 9 can be resolved automatically. This would further decrease the number of conflicts but is only applicable in specific scenarios.

Beside renaming, a further issue is that semistructured merge (much like structured merge) relies on structural information, so the revisions must be syntactically correct. Whereas it is best practice to commit only correct programs or documents, which has been confirmed by our study (we found only one Python project with a few problematic revisions), this is not a strict requirement of today's (unstructured) revision control systems. In such cases, the artifacts involved have to be parsed as plain text such that semistructured merge behaves exactly like unstructured merge. It is interesting to explore whether, in such cases, syntactically correct fragments can be represented by program structure trees and only the incorrect fragments as plain text.

In discussions at VAMOS'10 [6], some people asked how resolving ordering conflicts influences program comprehension. The background is that, in our approach, the merge tool decides the order (e.g., of method declarations in a class). An empirical assessment of this issue is well beyond the scope of this paper but an interesting avenue of further work. However, examinations of sample code make us believe that semistructured merge can aid program comprehension as it follows the structural boundaries of the artifacts involved in the merge process.

6. CONCLUSION

Both unstructured and structured revision control systems have strengths and weaknesses. The former are very general but cannot resolve certain kinds of conflicts. The latter are typically tailored to specific languages and thus can resolve conflicts better than the former. To reap the benefits of both worlds, we propose semistructured merge. Developers provide information on the artifact languages declaratively in the form of annotated grammars as well as in the form of pluggable conflict handlers. This way, a wide variety of different languages can be supported while taking advantage of the information provided during the merge process. We have implemented a tool for semistructured merge and plugged in support for Java, C#, and Python. In an empirical study, we found that semistructured merge can substantially reduce the number and size of conflicts that occur during merges, be it in terms of the number of conflicts (34±21 %), the number of conflicting lines of code (61±22 %), and the number of conflicting files (28±12 %). We also found situations in which semistructured merge increases the number of conflicts or the number of conflicting lines of code. But, in every case, either the number of conflicting lines of code or the number of conflicts decreases. In the presence of renaming, semistructured merge produces more but often smaller conflicts than unstructured merge. In general, semistructured merge is finer-grained than unstructured merge and disallows conflicts across class or method boundaries.

Interestingly, a combination of unstructured and semistructured merge can reduce the number of conflicts further (by 35 % compared to pure unstructured merge). In the future, we would like to explore whether it is possible to trace renaming and to use this information during the merge process.

7. ACKNOWLEDGMENTS

We are grateful to Don Batory and William Cook (who was also an author of the workshop paper on which this paper is based) for fruitful discussions on the potential of semistructured merge. We

thank Jens Dörre for preparing the grammars for FSTMERGE. Sven Apel's and Christian Lengauer's work is supported by the German Research Foundation (DFG – AP 206/2 and AP 206/4). Christian Kästner's work is supported by the European Research Council (ERC grant ScalPL #203099).

8. REFERENCES

[1] S. Apel and D. Hutchins. A Calculus for Uniform Feature Composition. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(5):Article 19, 2010.

[2] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *Proceedings of the International Conference on Model Transformation (ICMT)*, volume 5563 of *LNCS*, pages 4–19. Springer-Verlag, 2009.

[3] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Feature (De)composition in Functional Programming. In *Proceedings of the International Conference on Software Composition (SC)*, volume 5634 of *LNCS*, pages 9–26. Springer-Verlag, 2009.

[4] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE CS, 2009.

[5] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming*, 75(11):1022–1047, 2010.

[6] S. Apel, J. Liebig, C. Lengauer, C. Kästner, and W. Cook. Semistructured Merge in Revision Control Systems. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 13–19. University of Duisburg-Essen, 2010.

[7] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 161–170. IEEE CS, 2010.

[8] T. Apiwattanapong, A. Orso, and M. Harrold. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[9] V. Berzins. Software Merge: Semantics of Combining Changes to Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1875–1903, 1994.

[10] D. Binkley, S. Horwitz, and T. Reps. Program Integration for Languages with Procedure Calls. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(1):3–35, 1995.

[11] J. Buffenbarger. Syntactic Software Merging. In *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*, volume 1005 of *LNCS*, pages 153–172. Springer-Verlag, 1995.

[12] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[13] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 427–436. IEEE CS, 2007.

[14] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of Software Engineering Research on the Practice of Software Configuration Management. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(4):383–430, 2005.

[15] J. Grass. Cdiff: A Syntax Directed Differencer for C++ Programs. In *Proceedings of the USENIX C++ Conference*, pages 181–193. USENIX Association, 1992.

[16] D. Jackson and D. Ladd. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 243–252. IEEE CS, 1994.

[17] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering (TSE)*, 28(5):449–462, 2002.

[18] N. Niu, S. Easterbrook, and M. Sabetzadeh. A Category-theoretic Approach to Syntactic Software Merging. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 197–206. IEEE CS, 2005.

[19] Bryan O'Sullivan. Making Sense of Revision-Control Systems. *Communication of the ACM*, 52(9):56–62, 2009.

[20] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 183–194. ACM Press, 2005.

[21] B. Westfechtel. Structure-Oriented Merging of Revisions of Software Documents. In *Proceedings of the International Workshop on Software Configuration Management (SCM)*, pages 68–79. ACM Press, 1991.

[22] T. Zimmermann. Mining Workspace Updates in CVS. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, page 11. IEEE CS, 2007.