Semistructured Merge in Revision Control Systems

Sven Apel, Jörg Liebig, Christian Lengauer
Dept. of Informatics and Mathematics
University of Passau
{apel,joliebig,lengauer}@fim.uni-passau.de

Christian Kästner
School of Computer Science
University of Magdeburg
ckaestne@ovgu.de

William R. Cook
Dept. of Computer Sciences
University of Texas at Austin
wcook@cs.utexas.edu

***Abstract*—Revision control systems are a major means to manage versions and variants of today's software systems. An ongoing problem in these systems is how to resolve conflicts when merging 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 to inherit the strengths of both classes of systems: generality and expressiveness. The idea is to provide structural information of the underlying software artifacts in the form of annotated grammars, which is motivated by recent work on software product lines. This way, a wide variety of languages can be supported and the information provided can assist the resolution of conflicts. We have implemented a preliminary tool and report on our experience with merging Java artifacts. We believe that drawing a connection between revision control systems and product lines has benefits for both fields.**

I. Introduction

Revision control systems (a.k.a. version control systems) have a long tradition in software engineering [1], [2]. On the one hand, they are used in virtually every substantial software project in industry. On the other hand, they have also attracted much attention in academia. Revision control systems are a major means to manage versions and variants of today's software systems. 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.

In the 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 used widely in practice, since such systems are typically language-independent (i.e., they work with every software artifact that can be represented with text). Some widely used systems of this class are CVS[1], Subversion[2], Git[3], and Mercurial[4]. Henceforth, we call them *unstructured revision control systems*. A problem is that, when conflicts occur, the unstructured revision control system has no knowledge of the structure of the underlying software artifacts, which makes it difficult to resolve certain kinds of conflicts, as we will illustrate.

The second class is explored mainly in academia with the goal of solving the problems of unstructured revision control systems with the conflict resolution. The idea is to use the structure and semantics of the software artifacts being processed to resolve merge conflicts automatically [3]. These systems operate on abstract syntax trees or similar representations instead of on plain program text. A drawback is that, aiming at a particular language's syntax or semantics, they sacrifice language independence. Henceforth, we call these systems *structured revision control systems*.

Apparently, there is a trade-off between generality and expressiveness of revision control systems. 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 as many merge conflicts as possible automatically. Inspired by the trade-off between generality and expressiveness, we propose a new class of revision control systems, called *semistructured revision control systems*, that inherits the strengths but not the weaknesses of structured and unstructured revision control systems. The idea is to increase the amount of information a revision control system has at its disposal to resolve conflicts, while maintaining generality in the sense that many languages are supported. In particular, we concentrate on the merge process, so we speak of *semistructured merge*.

Our proposal is based on previous work on language-independent feature composition in software product line engineering [4], [5]. We noticed a strong similarity between software composition tools and software merging techniques used in revision control systems, which we exploit in our proposal. In a nutshell, we extend an existing feature composition tool infrastructure, called FeatureHouse, to enable it to merge different revisions of a software system based on the structure of the software artifacts involved. Users can plug new languages into FeatureHouse by providing a formal specification of their languages' syntax (i.e., the grammar) enriched with semantic information. While this approach is not entirely language-independent, it is still quite general in that new languages can be integrated easily by providing their grammars. If, for whatever reason, there is no grammar available for a certain language, a programmer can parse corresponding software artifacts line by line, which would be effectively the unstructured approach.

[1] http://www.cvshome.org/eng/
[2] http://subversion.tigris.org/
[3] http://git-scm.com/
[4] http://mercurial.selenic.com/

Base Program STACK

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

Fig. 1. A simple stack implementation in Java.

Revision TOP

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

Fig. 2. A revision of the stack implementation that adds method `top`.

Revision SIZE

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

Fig. 3. A revision of the stack implementation that adds method `size`.

Learning from product line engineering pays off in the development of revision control systems, as we will demonstrate. But also the reverse is interesting. Revision control systems are use widely in practice to manage versions and variants. We and others believe that drawing a connection between revision control systems and software product lines also provides insights for software product line engineers, especially with regard to real-world application scenarios [6].

In the remainder, we analyze the trade-off between generality and expressiveness of structured and unstructured merge. Based on the analysis, we derive our proposal of semistructured merge. Furthermore, we offer a preliminary tool that demonstrates the principal applicability of the approach, and we report on first experiences with it.

II. CONFLICTS IN REVISION CONTROL – BACKGROUND AND RELATED WORK

There is a large body of work on revision control systems [1], [2] and conflict resolution in software merging [3]. We concentrate on aspects relevant for our proposal. 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. By branching the development line, a programmer can create independent revisions, which can be changed and evolve in isolation (e.g., to add and test new features). Finally, independent revisions can be merged again 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. When two revisions have evolved independently, conflicts may occur while merging them. A major goal of research in this area is to empower revision control systems to resolve merge conflicts automatically [3]. First, we illustrate the problem of conflict resolution in unstructured merge. Then, we highlight some mechanisms of structured merge that enable them to resolve conflicts better than unstructured merge.

A. Unstructured Merge

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

Now, suppose a programmer would like to add a new feature TOP, but would like to develop the feature in its own branch, independently of the main branch (i.e., base program). To this end, the programmer creates a branch with a new revision TOP. Furthermore, suppose another programmer adds subsequently a feature `Size` directly to the main branch[5] by creating a corresponding revision of the base program. Figure 2 and Figure 3 present code for the two revisions, each of which add 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 features.

Merging the two branches involves merging the two revisions TOP and SIZE on the basis of the common ancestor, the base program STACK. This process is also called a *three-way merge* because it involves three programs or documents [2]. In our example, the merge process reports a conflict that cannot be resolved automatically with unstructured merge. Figure 4 illustrates the output of the Linux merge tool for this example. The figure shows that the merge process is not able to merge the two new methods `top` and `size` such that both can be present in the merged program.

This example is very simple but it illustrates already 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

[5]Note that the programmer could develop feature SIZE in any other branch but, for simplicity, we assume that the main branch is used.

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

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

Fig. 4. Output of the Linux merge tool when merging revision TOP and SIZE with the base program.

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

```
1  import java.util.LinkedList;
2  public class Stack<T> {
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 int size() {
11     return items.size();
12   }
13   public T pop() {
14     if(items.size() > 0) return items.removeFirst();
15     else return null;
16   }
17 }
```

Fig. 5. Merging TOP and SIZE without conflicts.

the new fragments. If the two revisions change or extend text in the same region, the system reports a conflict, i.e., 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 methods and that a merge of the two is actually straightforward, as we illustrate next.

Why is an unstructured revision control system not able to resolve the conflict that occurs when merging the revisions TOP and SIZE? As indicated before, an unstructured merge tool does not know that the two fragments implement Java methods whose order does not matter within a class declaration. If the tool knew that the base program and the two revisions are actually Java programs, then it would be able to solve the conflict automatically. There are actually two ways to resolve the conflict: include method `top` first and then method `size` (shown in Figure 5), or vice versa.

Revision SERIALIZABLE

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

Fig. 6. Revision that makes `Stack` objects serializable.

Revision FLUSHABLE

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

Fig. 7. Revision that makes `Stack` objects flushable.

B. Structured Merge

Figure 5 illustrates a very simple example of taking advantage of information on the syntax and semantics of the programs and revisions involved in the merge process. In the past, many tools have been proposed that leverage this kind of information to resolve as many conflicts as possible [3]. 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 [7], [8]. Researchers proposed a wide variety of structural comparison and merge tools including tools specific to Java [9] and C++ [10]. Some tools even consult additionally semantic information [11]–[13].

Let us illustrate the abilities of structured merge by a further example. Suppose we have the base stack implementation and we create two independent revisions, one to develop feature SERIALIZABLE that enables stack objects to be serialized and another to develop feature FLUSHABLE that allows programmers to flush the elements of the stack to a data stream. Figure 6 and Figure 7 depict excerpts of the two 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 two implements clauses of the two revisions. Figure 8 shows the conflicts as reported by the Linux merge tool.

A structured revision control system that knows that the base program and the revisions are written in Java is able to resolve the conflicts automatically and produces the desired result (i.e., the imports are placed one after the other and the implements clauses are concatenated), as shown in Figure 9.

Beside the conflicts we have seen so far, there are many more 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 header and a body, and the header consists of three parts. This information is useful when two revisions modify disjoint parts of the header.

$merge_{unstruct}$(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 Serializable {
9    private static final long serialVersionUID = 42;
10 =======
11 public class Stack<T> implements 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 }
```

Fig. 8. Output of the Linux merge tool merging revision SERIALIZABLE and FLUSHABLE with the base program.

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

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

Fig. 9. The desired result of merging revision SERIALIZABLE and FLUSHABLE with the base program.

C. Generality vs. Expressiveness

The previous discussion reveals that there is a trade-off between generality and expressiveness of revision control systems. Unstructured revision control systems are very general. They can be used with every kind of (textual) software artifact. However, they are not able to resolve conflicts that require knowledge on the language of the artifacts involved. Typically, a structured revision control system is 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 useless in a setting in which a software system consists of artifacts written in many different languages, most notably software product lines [4], [14].

The trade-off motivates us to explore the space between structured and unstructured revision control systems. Can we invent 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 a structured revision control system for every artifact type that occurs in a software project. 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; in times where people invent their own domain-specific languages and document formats, this approach is simply infeasible.

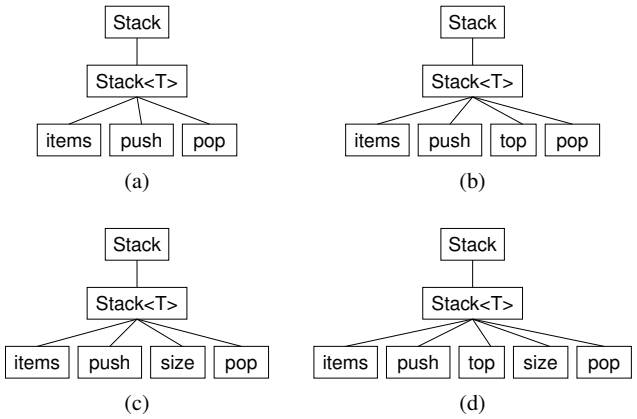

Fig. 10. Different revisions of the stack example represented as program structure trees.

III. SEMISTRUCTURED MERGE

The basic idea of semistructured revision control systems—which is much like in structured 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* (a.k.a. *feature structure tree*) [5]. In Figure 10 (a), we show a simplified program structure tree of the base program STACK, and, in Figure 10 (b) and Figure 10 (c), we show simplified program structure trees of 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 the structural 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 but abstracts from some details and represents them as plain text.

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 just by superimposing the trees, as shown in Figure 10 (d). Why does this algorithm work?[6] It works because the order of methods does not matter. 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

[6]Note that the structured merge tools of Westfechtel [7] and Buffenbarger [8] are not able to resolve this kind of conflict (personal communication with Westfechtel). However, in principle they would have enough information to do so.

do not have unique names). Even with all knowledge on the Java language, there are always cases in which we cannot say how to merge sequences of statements. This is the reason why we choose to represent methods as leaves and their statements as sole text content; in other languages, we may choose differently.

In the example of Figure 8, 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 concatenated.[7] But what do we do with program elements of which we do not know how to merge them, 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 cannot resolve it automatically—much like in unstructured merge.

So, semistructured merge is more expressive than unstructured merge because certain conflicts can be resolved automatically; but it is less expressive than truly structured merge because some content is treated as plain text. The question that arises is: Why not use structured merge altogether? The answer is: This way we loose more and more generality, as we explain next.

A. Balancing Generality and Expressiveness

The ability of semistructured merge to resolve the conflicts which we have discussed so far is based on the observation that the order of certain elements, e.g., of classes, interfaces, methods, imports, implements and throws list elements, and so on, does no matter. We call those conflicts *ordering conflicts*. A merge algorithm that just resolves ordering conflicts automatically is simpler to define than a full structural merge. A semistructured merge uses an *abstraction* of the structure of the document, where the abstraction has just enough information to identify ordered items and resolve conflicts.

Thus, our system consists of two parts: (1) a generic engine that knows how to identify and resolve ordering conflicts and (2) a small abstract specification—for each artifact type—of the program's or document's elements of which the order does not matter. The abstract specification of a document structure is given by an annotated grammar of the language. Most of the difficult 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. For example, the order of data type declarations in a Haskell program or of functions in a Python program does not matter.

To illustrate the role of annotations, consider the excerpt of a simplified Java grammar in Figure 11. It contains a set of production rules. For example, the rule `ClassDecl` defines the structure of classes containing fields (`FieldDecl`), constructors (`ClassConstr`), and methods (`MethodDecl`).

[7]Again, the structured merge tools of Westfechtel [7] and Buffenbarger [8] are not able to resolve this kind of conflict, but this should be possible in principle.

```
1  @FSTNonTerminal(name="{Type}")
2  ClassDecl : "class" Type ImplList "{"
3    (FieldDecl)* (ClassConstr)* (MethodDecl)*
4  "}";
5  ...
6  @FSTNonTerminal(name="ImplClause")
7  ImplList : "implements" @LIST Type ("," @LIST Type)*
8  @FSTTerminal(name="{<ID>}({ParamList})")
9  MethodDeclaration :
10   Type <ID> "(" (ParamList)? ")" "{"
11     (Statement)*;
12   "}";
13 @FSTTerminal(name="{TOSTRING}")
14 Type : ...
```

Fig. 11. An excerpt of a simplified Java grammar with semantic annotations.

Production rules may be annotated with `@FSTNonTerminal` and `@FSTTerminal`. The former annotation defines that (1) elements corresponding to the rule are represented as nodes in corresponding program structure tree, (2) there may be subnodes, and (3) the order of elements or nodes is arbitrary. In our example, we annotate the rule for class declarations with `@FSTNonTerminal`[8] because classes may contain further classes, methods, and so on, and the order of classes in a file or package may vary. The `@FSTTerminal` annotation is like the `@FSTNonTerminal` annotation except that subelements are represented as plain text. We annotate the rule for method declarations in this way because the order of methods may vary but their inner statements are represented by plain text, as explained before. A further interesting example is the rule for implements lists. This rule is annotated with `@FSTNonTerminal`, so the order of elements (i.e., type names) of an implements list may vary. However, for a parser, it is difficult to recognize that the elements form really a list, which is basically due to the grammar's treatment of the commas between the elements. The inner annotation `@LIST` passes exactly this information to the generated parser.

Beyond ordering conflicts we can imagine many ways to use annotations for conflict resolution. For example, we could use annotations to specify how the parts of a `for` loop header are merged. In this case, the order of the parts matters, so the annotations would have to be of a different kind. In further work, we will explore the potential of our approach to resolve other kinds of conflicts.

As we have illustrated, an annotated grammar contains sufficient information to guide a language-independent 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, so the tool is not entirely language-independent. But 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. Actually, we do not even need the entire grammar of a language, but only the part that is concerned with elements whose order is flexible. We have

[8]The annotation parameter `name` is used to assign a name to the corresponding nodes in the program structure tree.

been quite successful using such a mechanism in feature composition in software product line engineering [4] and we expect to reproduce the success of applying such a mechanism in revision control systems.

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 solely on the basis of providing an annotated grammar, which needs to be done only once per language. If, for whatever reason, there is no information available on a given language, semistructured merge behaves exactly like unstructured merge, parsing the corresponding software artifact line by line.

IV. Implementation and Experience

We have implemented a first prototype of a semistructured merge tool, called FSTMerge, which is able to resolve ordering conflicts.[9] FSTMerge takes advantage of our existing tool infrastructure FeatureHouse, as illustrated in Figure 12. 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 proprietary format, called FeatureBNF, of which we have shown already an example in Figure 11. Using a grammar written in FeatureBNF, FSTGenerator generates an LL(k) parser (which produces program structure trees) and a corresponding pretty printer, which are then integrated into FSTMerge. After the generation step, FSTMerge proceeds as follows: (1) the generated parser receives the base program and two revisions written in the target language and produces for each program a program structure tree; (2) FSTMerge performs the semistructured merge as explained before (the trees are superimposed and a conventional unstructured merge is applied to the leaves); (3) the generated pretty printer writes the merged revisions to disk.

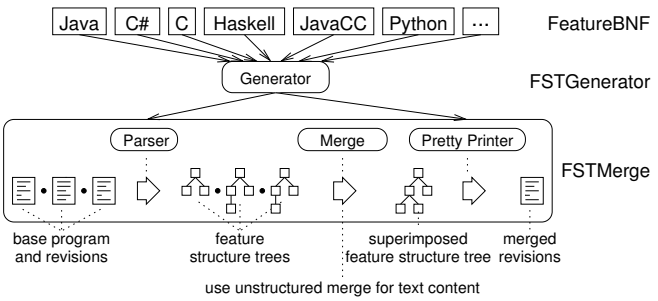

Fig. 12. The architecture of FeatureHouse.

So far, we have used FSTMerge only with Java programs and we concentrated on ordering conflicts (including merging classes containing methods, fields, implements lists). But, due to our experience with FeatureHouse in software product line engineering [4], [5], [15]–[18], we expect that integrating further languages is very easy. In fact, we have already developed the annotated grammars of several further languages including C, C#, JavaCC, and Haskell—what is missing are case studies. The more interesting issue is whether semistructured merge can play to its strengths in real software projects. It is clear that semistructured merge is able to resolve more conflicts than unstructured merge. But how frequent are such conflicts? For example, how often are methods added independently to the same region? How often are implements lists changed? Currently, we cannot provide answers. Although there is some evidence that revisions often involve additions of larger structures such as entire functions [19], we need a substantial set of data to answer the question definitely. From the theoretical point of view, semistructured merge is very interesting, not least because, by means of playing with annotations, we can adjust the way the merge tool works. However, the impact on practical revision control remains to be evaluated. A first step is to analyze the kind and frequencies of conflicts in different software projects incorporating different kinds of software artifacts.

[9] FSTMerge and some examples can be downloaded with the FeatureHouse distribution: http://www.fosd.de/fh/

V. Conclusion and Open Issues

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 can thus resolve conflicts better than the former. To profit from both worlds, we have proposed semistructured merge, which is inspired by our previous work on software product lines. Developers provide information on the artifact languages in the form of annotated grammars. This way, a wide variety of different languages can be supported while taking advantage of the provided information during the merge process. We have implemented a preliminary tool and plugged in support for the Java language. Whereas integrating further languages is straightforward, it is interesting to explore whether semistructured merge can play to its strengths in practical software engineering. Finally, it is interesting to study the commonalities and differences of software product lines and revision control systems. We have taken a first step and we believe that, in the future, both fields will converge.

We see three interesting open issues of our approach. The first 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, 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 syntacticly correct fragments can be represented by program structure trees and only the incorrect fragments as plain text.

A further issue is the role of refactorings. So far we have not addressed changes like the renaming of methods or classes. For example, in semistructured merge, a rename method refactoring would result in two different methods (the original method and the renamed method), without reporting a conflict. It is debatable if this is the desired behavior. On the other hand, we believe that structural information of whatever kind is of a great value in the presence of refactoring. One can even imagine to tune the kind of information that is passed to the merge tool, e.g., information on references between program elements instead of ordering information. We will explore this issue in further work.

Finally, it would interesting to explore how type information can be used to resolve conflicts. The problem is that type systems are typically tailored to specific languages and thus would undermine generality. However, researchers begin to think about cross-language and language-independent type systems [20]–[22]. In the future, it may be possible to use such a type system for conflict resolution in semistructured revision control systems.

Acknowledgments

We thank Don Batory for fruitful discussions on the potential of semistructured merge. This work has been supported in part by the German Research Foundation (DFG), project number AP 206/2-1.

References

[1] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 232–282, 1998.

[2] B. O'Sullivan, "Making Sense of Revision-Control Systems," *Communications of the ACM (CACM)*, vol. 52, no. 9, pp. 56–62, 2009.

[3] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 5, pp. 449–462, 2002.

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

[5] S. Apel and C. Lengauer, "Superimposition: A Language-Independent Approach to Software Composition," in *Proceedings of the International Symposium on Software Composition (SC)*, ser. Lecture Notes in Computer Science, vol. 4954. Springer-Verlag, 2008, pp. 20–35.

[6] M. Staples and D. Hill, "Experiences Adopting Software Product Line Development without a Product Line Architecture," in *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, 2004, pp. 176–183.

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

[8] J. Buffenbarger, "Syntactic Software Merging," in *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*, ser. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, 1995, pp. 153–172.

[9] T. Apiwattanapong, A. Orso, and M. Harrold, "JDiff: A Differencing Technique and Tool for Object-Oriented Programs," *Automated Software Engineering*, vol. 14, no. 1, pp. 3–36, 2007.

[10] J. Grass, "Cdiff: A Syntax Directed Differencer for C++ Programs," in *Proceedings of the USENIX C++ Conference*. USENIX Association, 1992, pp. 181–193.

[11] V. Berzins, "Software Merge: Semantics of Combining Changes to Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1875–1903, 1994.

[12] 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)*. IEEE Computer Society, 1994, pp. 243–252.

[13] D. Binkley, S. Horwitz, and T. Reps, "Program Integration for Languages with Procedure Calls," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 4, no. 1, pp. 3–35, 1995.

[14] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 6, pp. 355–371, 2004.

[15] S. Apel, C. Lengauer, B. Möller, and C. Kästner, "An Algebra for Features and Feature Composition," in *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*, ser. Lecture Notes in Computer Science, vol. 5140. Springer-Verlag, 2008, pp. 36–50.

[16] 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)*, ser. Lecture Notes in Computer Science, vol. 5563. Springer-Verlag, 2009, pp. 4–19.

[17] 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)*, ser. Lecture Notes in Computer Science, vol. 5634. Springer-Verlag, 2009, pp. 9–26.

[18] S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner, "An Algebra for Feature-Oriented Software Development," Department of Informatics and Mathematics, University of Passau, Tech. Rep. MIP-0706, 2007.

[19] 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)*. ACM Press, 2005, pp. 183–194.

[20] M. Grechanik, D. Batory, and D. Perry, "Design of Large-Scale Polylingual Systems," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2004, pp. 357–366.

[21] S. Apel and D. Hutchins, "An Overview of the gDeep Calculus," Department of Informatics and Mathematics, University of Passau, Tech. Rep. MIP-0712, 2007.

[22] S. Apel and D. Hutchins, "A Calculus for Uniform Feature Composition," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2010.