Master Essay

Eirik Halvard Sæther

February 21, 2020

Contents

Chapter 1

Notes

1.1 Reference examples

- Precise Version Control of Trees with Line-Based Version Control Systems (Asenov et al. 2017)

- A three-way merge for XML documents (Lindholm 2004). See Section 2.8.1

- Fine-grained and accurate source code differencing (Falleri et al. 2014)

- Using versioned trees, change detection and node identity for three-way XML merging (Thao and Munson 2019)

- Generic Diff3 for algebraic datatypes (Vassena 2016)

- A State-of-the-Art Survey on Software Merging (Mens 2002)

Chapter 2

Introduction

2.1 Unstructured Paragraphs

2.1.1 Version Control Systems

Why the need for VCS

Software configuration mechanisms is the discipline of managing the evolution of large and complex software systems (61 in (Mens 2002))

Version control mechanisms (12 in mens), are used to deal with the evolution of software products. These mechanisms include ways to deal with having multiple, parallel versions of the software simultaneously. Techniques like *software merging* are used to keep consistency and unify different versions by automatically or semi-automatically deriving merged versions of the different parallel versions.

Two-way vs three-way merging

When merging different versions of a piece of software, we differentiate between *two-way* and *three-way* merging. Two-way merging merges the two versions without taking a common ancestor into account. Three-way merging on the other hand, uses a common ancestor as a reference point, to know how the different versions were changed. The latter technique is more powerful and produces more accurate merges, because the merge will know extra information. Using two-way merge, if a piece was removed from one version, the two-way merge would not know whether the piece was removed in the first file or added in the second. This is not a problem in three-way merge, since we know from the common ancestor whether there was a deletion or addition. In this thesis, we will focus on three-way merging, since it will produce better results.

Representations in the merge

Merge tools can be further categorized in how they represent the data they are merging. As proposed in (Mens 2002), a way of categorizing them are *textual*, *syntactic* and *semantic* merging. The most common technique is definitely textual merging, where the software is treated as a flat text file, and merging is done without consideration of the intended structure or semantics of the text file.

Textual merging Textual merging views the software artifacts as unstructured text files. There exist several granularities of what is considered one unit, but *line based merging* is probably the most common textual merge. Line based merging techniques computes the difference between files by comparing equality over the lines. This has several implications, like adding a single space after a line is considered a deletion of the old line and addition of the new. This coarse granularity often leads to unnecessary and confusing conflicts. Changing the indentation or other formatting differences often lead to unnecessary conflicts.

To exemplify this, consider the two versions of a python file, Listing 1 and Listing 2. The second version simply wrapped the content of the function in an if-statement that checks for input sanity. Using a standard textual, line based differencing tool like the Unix' *diff*-tool [[TODO: ref, see tom for specifics]], we are able calculate the difference between the two files by calculating the longest common subsequence. As seen in the result Listing 3, difference between the two are confusing and inaccurate. Conceptually, the difference was that the second version wrapped the block in a if-statement. Due to the coarse grained line based differencing and the disregard of structure and semantics, the algorithm reported that the whole block was deleted, and the same block wrapped in an if was inserted.

```python
def some_function(n):
  sum = 0
  for i in range(0, n):
    sum += i
some_function(5)
```

Listing 1: Code diff 1

As discussed, text-based merge techniques often provide inferior results, however, they have several advantages. Because of the algorithms focus on generality, accuracy and efficiency, these types of algorithms are widely adopted. The algorithm is general enough to work well for different programming languages, documentation, markup files, configuration files, etc. Some measurements performed on three-way,

```
def some_function(n):
  if isinstance(n, int):
    sum = 0
    for i in range(0, n):
      sum += i
some_function(5)
```

Listing 2: Code diff 2

```
<    sum = 0
<    for i in range(0, n):
<      sum += i
<    print(sum)
---
>    if isinstance(n, int):
>      sum = 0
>      for i in range(0, n):
>        sum += i
>      print(sum)
```

Listing 3: Resulting code diff

textual, line-based merge techniques in industrial case studies showed that about 90 percent of the changed files could be merged automatically [[TODO: ref 49 in mens]]. Other tools can complement the merge algorithm in avoiding or resolving conflicts. Formatters can make sure things like indentation and whitespace are uniformly handled, to avoid unnecessary conflicts. Compilers can help in resolving conflicts arising from things like renaming, where one version renames a variables, while another version introduces new lines referencing the old variable.

Syntactic Merging Syntactic merging [[TODO: ref, 10, mens]] differs from textual merging in that it considers the syntax of the artifact it is merging. This makes it more powerful, because depending on the syntactic structure of the artifact, the merger can ignore certain aspects, like whitespace or code comments. Syntactic merge techniques can represent the software artifacts in a better data structure than just flat text files, like a tree or a graph. In example, representing the Python program from Listing 1 and Listing 2 as a parse tree or abstract syntax tree, we can avoid merge conflicts. [[TODO: maybe a nice drawing or something, with explanation]].

The granularity of the merger is still relevant, because we sometimes want to report a conflict even though the versions can be automatically merged. Consider the following example. $n < x$ is changed to $n \leq x$ in

one version, and to $n < x + 1$ in another. Too fine grained granularity may cause this to be merged conflict free as $n \leq x + 1$. The merge can be done automatically and conflict free, but here we want to report a warning or conflict, because the merge might lead to logical errors.

Semantic Merging   While syntactic merging is more powerful than its textual counterpart, there are still conflicts that go unnoticed.   The syntactical mergers can detect conflicts explicitly encoded in the tree structure of the software artifact, however, there often exist implicit, cross-tree constraints in the software. An example of such a constraint references to a variable. The variable references in the code are often semantically tied to the definition of the variable, where the name and scope implicitly notes the cross tree reference to the definition.

Consider the following simple program: `var i; i = 10;`. If one version changes the name of the variable: `var num; num = 10;`, and another version adds a statement referencing the variable: `var i; i = 10; print(i)`. Syntactic or textual mergers would not notice the conflict arising due to the implicit cross-tree constraints regarding the variable references, and merge the versions conflict-free with the following, syntactically valid result: `var num; num = 10; print(i)`

2.1.2   Feature Models

Introducing feature models. What they are, why they are useful. One part is allowing different collaborators. Important for autonomy, etc.

Something about diff, three-way merge and working on line granularity. Very inaccurate, not adequate. Maybe a small example.

possible a paragraph about different version control mechanisms and techniques. Pessimistic and optimistic approaches. (Mens 2002)

Introduce a method of merging two feature models derived from the same feature model using a three-way merge algorithm inspired from the text-granularity algorithm and other three-way merge algorithms working on tree-structures like XML and ASTs.

These solutions are not adequate, because we want to create an algorithm that ensures soundness and takes the semantics of feature models into account when merging. This task is not trivial, and the use cases of the developers will need to be considered when deciding which conflicts to automatically merge, warn the user, etc. The algorithm will utilize properties about feature models when merging, like the fact that the nodes are unordered, and each node is uniquely identified by their id.

This task includes defining the semantics of feature models, defining what a sound model is. The three-way merge algorithm will assume three sound feature models as input, and compute the edit script between the base and the two derived feature models. The two edit scripts will then be

combined to a graph that represents what changes to be done to the base, including possible conflicts and dependencies.

2.2 Motivation

TODO

2.3 Objective (with solution requirements)

TODO

2.4 Difficulties/problems/challenges

TODO

2.5 Say why other solutions are not adequate (very short)

TODO

2.6 Describe the solution (very short)

TODO

2.7 Roadmap of the thesis

TODO

2.8 Related Work

2.8.1 A Three-way Merge for XML Documents

Lindholm describes a method of merging XML documents. The algorithm is a three-way merge taking a base XML file, T_0, and two files T_1 and T_2, derived and changed from the base file independently. The XML documents are ordered, labeled trees, and merging the trees takes this into consideration. The algorithm relies on computing *edit scripts* between T_0 and T', where T' is either T_1 or T_2. The edit scripts consist of predefined

operations, which are attribute updates and insert, move and delete for XML elements.

One of the important factors of the data structure, is that the nodes are labeled with unique IDs. This makes merging significantly easier.

To study different use-cases (user stories) of XML merging, they manually merged some examples, to see how real world users would handle some tricky cases. Some rules for merging were devised, including:

- Not possible to move a node in T_1 that are being updated in T_2

- Guards, aka node contexts are there to handle merging changes originating too close to each other. This isn't technically a merge conflict, but a potential semantic issue.

- Normal stuff. Added or removed nodes should be reflected in the merge. Parent relation should be updated

2.9 A State-of-the-Art Survey on Software Merging

Bibliography

Asenov, Dimitar et al. (2017). "Precise Version Control of Trees with Line-Based Version Control Systems". In: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pp. 152–169. DOI: 10.1007/978-3-662-54494-5_9. URL: https://doi.org/10.1007/978-3-662-54494-5%5C_9.

Falleri, Jean-Rémy et al. (2014). "Fine-grained and accurate source code differencing". In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pp. 313–324. DOI: 10.1145/2642937.2642982. URL: https://doi.org/10.1145/2642937.2642982.

Lindholm, Tancred (2004). "A three-way merge for XML documents". In: *Proceedings of the 2004 ACM Symposium on Document Engineering, Milwaukee, Wisconsin, USA, October 28-30, 2004*, pp. 1–10. DOI: 10.1145/1030397.1030399. URL: https://doi.org/10.1145/1030397.1030399.

Mens, Tom (2002). "A State-of-the-Art Survey on Software Merging". In: *IEEE Trans. Software Eng.* 28.5, pp. 449–462. DOI: 10.1109/TSE.2002.1000449. URL: https://doi.org/10.1109/TSE.2002.1000449.

Thao, Cheng and Ethan V. Munson (2019). "Using versioned trees, change detection and node identity for three-way XML merging". In: *Computer Science - R&D* 34.1, pp. 3–16. DOI: 10.1007/s00450-013-0253-5. URL: https://doi.org/10.1007/s00450-013-0253-5.

Vassena, Marco (2016). "Generic Diff3 for algebraic datatypes". In: *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, pp. 62–71. DOI: 10.1145/2976022.2976026. URL: https://doi.org/10.1145/2976022.2976026.