Master Essay

Eirik Halvard Sæther

April 17, 2020

Contents

Chapter 1

Notes

1.1 Reference examples

- Precise Version Control of Trees with Line-Based Version Control Systems [1]
- A three-way merge for XML documents [3]. See Section 3.3.1
- Fine-grained and accurate source code differencing [2]
- Using versioned trees, change detection and node identity for three-way XML merging [5]
- Generic Diff3 for algebraic datatypes [6]
- A State-of-the-Art Survey on Software Merging [4]

Chapter 2

Introduction

2.1 Motivation

Feature models are designed to help engineers cope with the long term evolution of software. Designing the software is a iterative, dynamic process, which is subject to change. Having several engineers working in parallel on the same feature model can be beneficial. This requires good tools for handling several engineers working, changing and synchronizing the feature model.

2.2 Objective (with solution requirements)

To allow several engineers working in parallel, the goal of this thesis is to design and implement a method of synchronizing and merging the changes from different collaborators on the same feature model. Since the changes from the collaborators were devised from a common feature model, the algorithm should take this common ancestor into consideration when merging, to produce quality results. The algorithm should always produce a syntactically and semantically valid feature model, to ensure a sound, valid feature model as a result. In cases where the merging of the different revisions are not trivial, the method should give ways of resolving conflicts by consulting the user.

2.3 Difficulties/problems/challenges

TODO

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

[[TODO: maybe ref linholm or json merger, how this is not good enough as well. also use textual/syntacic/etc]] While there are several tools to handle three-way merging, they do not produce adequate results to solve the problem. Tools such as diff [[TODO: github, unix, ref]] are one of the most used tools for the task, due to their generality and efficiency. However, several of the more implicit semantic requirements of feature models could go unnoticed in the merge result, producing a invalid feature model without the algorithm giving any warnings. There are also several cases where other, more general algorithms would produce a conflict. When you move a feature to another part of the tree, and then modify the name, etc, general algorithms would not know if you deleted a feature and created a new, or if you moved and changed the same feature. However, due to the well defined semantics of feature models, we know that every feature has a unique identifier, which can be leveraged by the merge algorithm.

2.5 Describe the solution (very short)

TODO

2.6 Roadmap of the thesis

TODO

Chapter 3

Background

3.1 Version Control Systems

Software configuration mechanisms is the discipline of managing the evolution of large and complex software systems [[TODO: (61 in [4])]] *Version control mechanisms* [[TODO: (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.

Mens [4] categorizes and describes different aspect of version control systems and software merging techniques. Two-way and three-way merging differentiates between how many versions of the artifact you are comparing. Different representations of the merge artifact can be categorized in textual, syntactic, semantic or structural merging. State-based merge techniques uses delta algorithms to compute differences between revisions while change-based techniques keeps track of the exact operations that were performed between the revisions.

3.1.1 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.

3.1.2 Representations in the merge

Merge tools can be further categorized in how they represent the data they are merging. As proposed in [4], 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.

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

Listing 1: Code diff 1

```
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

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, 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 is 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)`.

Semantic mergers takes these kinds of conflicts into consideration while merging. Using *Graph-based* or *context-sensitive* merge techniques, we can model such cross tree constraints, by linking definitions and invocations with edges in the graph. However, in some cases, such *static semantic* merge techniques are not sufficient. Some changes cannot generally be detected statically, and may need to rely on the runtime semantics.

3.2 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. [4]

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.

3.3 Related Work

3.3.1 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

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

Chapter 4

Solution

4.1 Notation

Throughout this thesis, I will use the following variable naming convention. Let F_b be the initial base feature model, while F_1 and F_2 be two different feature models, derived from model F_b. To note either of the two derived ones, we use the notation F_d. We call the resulting feature model after the merge algorithm is performed F_m.

The edit script, which is the calculated list of operations to go from F_b to one of the derived models F_d, we note as ε. If we need to be specific to which of the derived models were computed, we note this with ε_1 and ε_2.

4.2 Feature model semantics

Definitions of feature models, groups, etc. Taken from the paper stuff [[TODO: write]]

4.3 Change detection

One step of computing F_m from F_b, F_1 and F_2, is computing the two edit scripts ε_1 and ε_2. The edit scripts are computed by a directed delta algorithm [4]. The delta algorithm has three distinct phases. (1) The algorithm will compute the difference between F_b and F_d, and produce a set of operations (See the operation table 4.1) needed to get from the base to the derived model. (2) The necessary operations and dependencies between the operations are modeled in a dependency graph. This is due to certain operations needing to be performed before others, i.e. every child has to be removed before you can remove the parent. (3) A topological sorting is performed on the graph to compute an ordered list of operations. When performing the operations in order starting with F_b, the resulting feature model will be equal to the actual derived one. This eliminates the

need to track and log the operations when they are being performed by a user. [[TODO: not actually what is going to happen I think]]

Table 4.1: List of operations
[[TODO: make as table]]

- Add feature
- Add group
- Move feature to group
- Move group to feature
- Delete feature
- Delete group
- Change feature
- Change group

4.3.1 Dependencies between merge changes

Cycle Detection

Based on the assumption that the base F_b, and the derived feature models, F_1 and F_2 are valid and sound feature models, we know that they contain no cycles. However, when synthesising the merged model, cycles can occur if both edit scripts contain move operations. If ε_1 contains `MoveFeature(Fid, Gid)`, and ε_2 contains `MoveFeature(Fid', Gid')`, we need to ensure that integrating the changes doesn't produce any cycles. If `Fid` is moved to `Gid`, problems can occur if `Fid'` now becomes the new ancestor of `Fid` and `Gid'` is a subgroup of `Fid`.

Bibliography

[1] Dimitar Asenov et al. “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*. 2017, 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.

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

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

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

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

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