Three Way Merge for Feature Model Evolution Plans

Eirik Halvard Sæther

Thesis submitted for the degree of
Master in Informatics: Programming and Systems Architecture

60 credits

Department of Informatics

Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Three Way Merge for Feature Model Evolution Plans

Eirik Halvard Sæther

Acknowledgements

Thanks to Ida

Thanks to Ingrid and Crystal

Thanks to Germans, everyone on the LTEP project for input etc.

Thanks to ifi, all the people i have met

Thanks to friends and family

Abstract

[[TODO: write abstract]]

Feature Model Evolution Plans is intended to help ease the development of software product lines (SPLs). Feature Models allow software engineers to explicitly encode the similarities and differences of an SPL. However, due to the changing nature of an SPL, Evolution Plans allows for representing the *evolution* of a feature model, not just the feature model as a single point in time.

Evolution planning of an SPL is often a dynamic, changing process, due to changing demands of the focus of development. The evolution planning is often not just done by a single engineer, but multiple engineers, working separately and independent of each other. Due to these factors, the need to unify and synchronize the changes the evolution plan emerges.

In this thesis, we develop a merge tool for Feature Model Evolution Plans. The core of the tool is a three-way merge algorithm. Given two different versions of an evolution plan, together with the common evolution plan they were derived from, the merge algorithm will attempt to merge all the different changes from both versions. If the merges are unifiable, the algorithm will succeed and yield the merged result containing the changes from both versions. However, if the changes are conflicting in any way, breaking the structure or semantics of evolution plans, the algorithm will stop, telling the user the reason of failure.

Contents

List of Figures

List of Tables

Preface

[[TODO: write better and more]] something about the LTEP project

something about summer project?

Chapter 1

Introduction

introdusere problemstillingen. si hva spl, fm, ep er på ett høynivå

i introduksjon, fortelle hva feature models er, hva det kan brukes til. ikke tekniske detaljer! engineering practice, software family. referanser. styrker. videre inn i planlegging, vanskelig. flere teams som må planlegge. hvordan hjelpe dette. gjenbruke kode. scope: her modellerer jeg bare planlegging. hvordan min oppgave skal leses.

1.1 Motivation

1.2 Problem Statement

1.3 Research Questions

forskningsspørsmål/mål. hva skal jeg undersøke/belyse. motivere, hvorfor gjør jeg det jeg gjør konkrete spørsmål. how? what? in what way? type spm hvordan formalisere endring? hva er en god effektiv implementasjon? fornuftig algoritme som tar hensyn til semantikk og struktur? identifisere hva som er utfordringen. 3 stk ca. for å addressere spm så må jeg se på følgende:

hvor er de store utfordringene? finne riktig representasjon? rekkefølge på operasjoner? hva gjør problemstillingen vrien?

1.4 Contributions

what i have done

including formalizing and implementing a 3wm algo that preserves soundness implemented the algorithm in haskell created an entire program with a command line interface, that handles different formats, reads/writes to JSON files, logging, etc. Created a frontend in Elm for a dynamic, actual presentation of the input and results of the program Created examples and tests, checking that the program behaves as intended.

[[TODO: WRITE]]

1.5 Chapter Overview

Chapter 2 something about background

Chapter 4 something about the bigboy algo

[[TODO: WRITE]]

1.6 Project Source Code

All the source code from the master thesis can be found on Github[1].

[1]https://github.com/eirikhalvard/master-thesis

Chapter 2

Background

bakgrunn er ting vi vet i dag om spl. ikke blande inn min contribution i bakgrunn. tydelig skille. i bakgrunnseksjon: diskutere litt hvordan forskjellige merge teknikker har fordeler/ulemper.

2.1 Software product lines

A software product line (SPL) is a family of closely related software systems. These systems will often have several features in common, as well as variations that makes each piece of software unique. SPLs are used to make highly configurable systems, where each product in the SPL, called a *variant*, is defined by the combination of features chosen.

Software product line engineering is a discipline for efficiently developing such families of software systems. Instead of maintaining potentially hundreds of different software artifacts, these engineering methods have ways of capitalizing on the similarities and differences between each variant. The number of variants are subject to combinatorial explosion, with additions of new features may double the amount of variants. Developing software product lines can be very time efficient, because you can maintain one code base, instead of one code base per variant. This simplifies additions of features or bug fixes greatly.

2.2 Feature Models

All possible variants of a software product line can be defined in terms of a *feature model*. A feature model is a tree structure of features and groups. Features can be mandatory or optional, and will contain zero or more groups. Each group has a set of features. A group (of features) can have different types. For example, in an `AND` group, all the features has to be chosen.

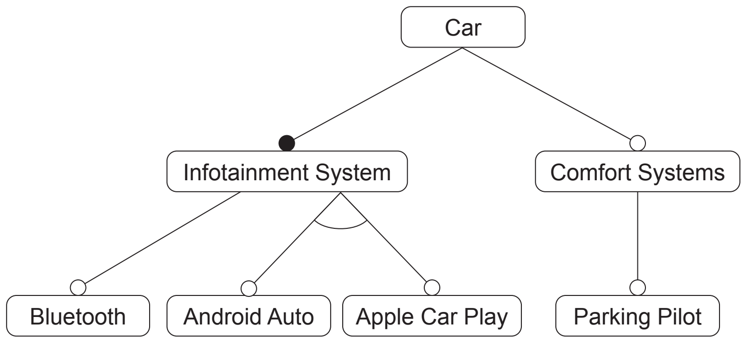


Figure 2.1: Example feature model

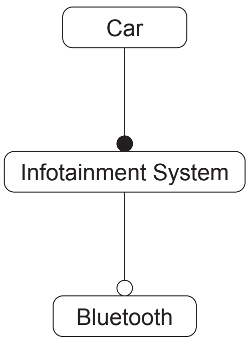
A visual representation of a feature model can be seen in Figure 2.1. The small dot above `Infotainment System` indicates that the feature is mandatory, where as the white dot above `Comfort Systems` represents an optional feature. Each feature (except the root) is in a group. The `Infotainment System` feature is in a singleton group below `Car`. The features `Android Auto` and `Apple Car Play` are in a `XOR` group, indicated by the arch between the features. This represents that each valid variant has to choose between one of the two (but not both).

2.3 Evolution planning

Feature models let engineers capture all variants of the current software product line, but sometimes it can be beneficial to model future or past versions as well. Planning for the long term evolution of the product line can be important in managing the complexity that comes with large software systems. Developing these kinds of systems typically involves many engineers, managers or other stakeholders, and managing when certain changes, additions or deprecations are implemented can be complex and confusing without suitable tools. Changing the SPL potentially influences many configurations, which might conflict with the stakeholders requirements.

SPL evolution is a major challenge in SPL engineering as many stakeholders are involved, many requirements exist, and changing the SPL potentially influences many configurations. Thus, it is paramount to thoroughly plan SPL evolution in advance, e.g., to perform analyses and to have enough time for implementing new or adapted features.

Evolution plans lets us model a sequence of feature models, which represents the current and all planned future versions of the feature model. Each feature model represents the product line in a point in time, which could have varying validity, from a week from now to a year. Since the next feature model is derived from the previous one, we can represent the evolution plan as an initial feature model, as well as a sequence of *points*, where each point is a set of operations to perform on the previous feature model to achieve the current one. The operations vary from changing, adding or deleting features or groups from the feature model.



At time 1:

add an XOR group to Infotainment System.
add feature Android Auto to the Infotainment System XOR group
add feature Car Play to the Infotainment System XOR group

At time 2:

add feature Comfort Systems to the Car AND group
add an AND group to Comfort Systems
add feature Parking Pilot to the Comfort Systems AND group

Figure 2.2: An example evolution plan

An example of an evolution plan can be seen in Figure 2.2. The initial feature model contains three features, and two time points are added. At time 1, a group and two features are added, and at time 2, another group and two features are added. The evolution plan can derive three feature models, the initial, and the two at time 1 and 2. Performing all the operations results in a feature model that is equal to the one in Figure 2.1 on the preceding page

2.4 Version Control Systems

Software configuration mechanisms is the discipline of managing the evolution of large and complex software systems [5]. *Version control mechanisms* 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 [3] 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.

2.4.1 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 from the common ancestor.

To illustrate the difference, consider the following program: `print(a); print(b); print(a + b)`, and two different versions derived from the base program, (1) `print(a); print(b); print(a+b); print("new line")`, (2) `print(b); print(a + b)`. Since a three-way merger uses the base program as a reference point, it will notice that derived version 1 added one statement, while version two deleted one. The three-way merger will then merge successfully without conflict with the following result: `print(b); print(a + b); print("new line")`. However, a two-way merger does not use the base program the different versions were derived from, and can not deduce whether `print(a)` were added in version 1 or deleted in version 2, thus raising a conflict. The same ambiguity occurs with the added statement `print("new line")`.

2.4.2 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 program, 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 [2], we are able to calculate the difference between the two files by calculating the longest common subsequence. As seen in the result (Listing 3 on the next page), difference between the two are confusing and inaccurate. Conceptually, the difference is 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 reports that the whole block is deleted, and the same block wrapped in an if is inserted.

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

Listing 1: Code diff 1

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

Listing 2: Code diff 2

As discussed, text-based merge techniques often provide inferior results, however, they have several advantages in terms of efficiency and general-

```
<   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

ity. 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 [4]. 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.

2.4.3 Syntactic Merging

Syntactic merging [1] 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 on the preceding page and Listing 2 on the previous page as a parse tree or abstract syntax tree, we can avoid merge conflicts.

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.

2.4.4 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.

2.5 Haskell and Algebraic Data Types

Chapter 3

Formal Semantics of Feature Model Evolution Plans

1. vise sensor at det er publiserbar vitenskaplig arbeid. 2. novelty. det jeg har gjort er ett faktisk nytt vitenskaplig bidrag. sitere paper vi har fått inn splc. arbeid er basert på sunn plan. det skal ikke oppfattes som to oppgaver. fortelle om hvordan vi formelt har en sunn plan. cornerstone i mitt arbeid med merging, fordi det bygger på en plan som er sunn. kan si dette arbeidet er publisert. 3 og 4 til min oppgave. ikke nevn tidligere arbeid eksplisitt (det er en del av selve oppgaven) kan snakke om operasjonene, hva de betyr på høynivå. referere til artikkel for detaljer.

Chapter 4

Three Way Merge Algorithm

4.1 Algorithm Overview

4.1.1 Three-Way Merging of Evolution Plans

The three-way merge algorithm for feature model evolution plans will take two different versions of an evolution plan, *version 1* and *version 2*, and attempt to merge the evolution plans into a single plan. In order to do so, a third evolution plan has to be provided, which is the common evolution plan they were derived from. The common evolution plan, called *base*, will implicitly provide information about what things were added, removed and changed in each of the derived evolution plans.

4.1.2 Soundness Assumption

The three-way merge algorithm will assume that the three evolution plans provided are sound. By assuming the soundness of the plans, the algorithm can leverage this to create a better merge result. But more importantly, the assumption is based around the fact that there is no point in merging an evolution plan you know violates soundness in some way.

4.1.3 Algorithm Phases

In order to merge the different versions of the evolution plan, the algorithm is separated into several distinct phases. The different steps and phases of the algorithm can be seen in Figure 4.1.

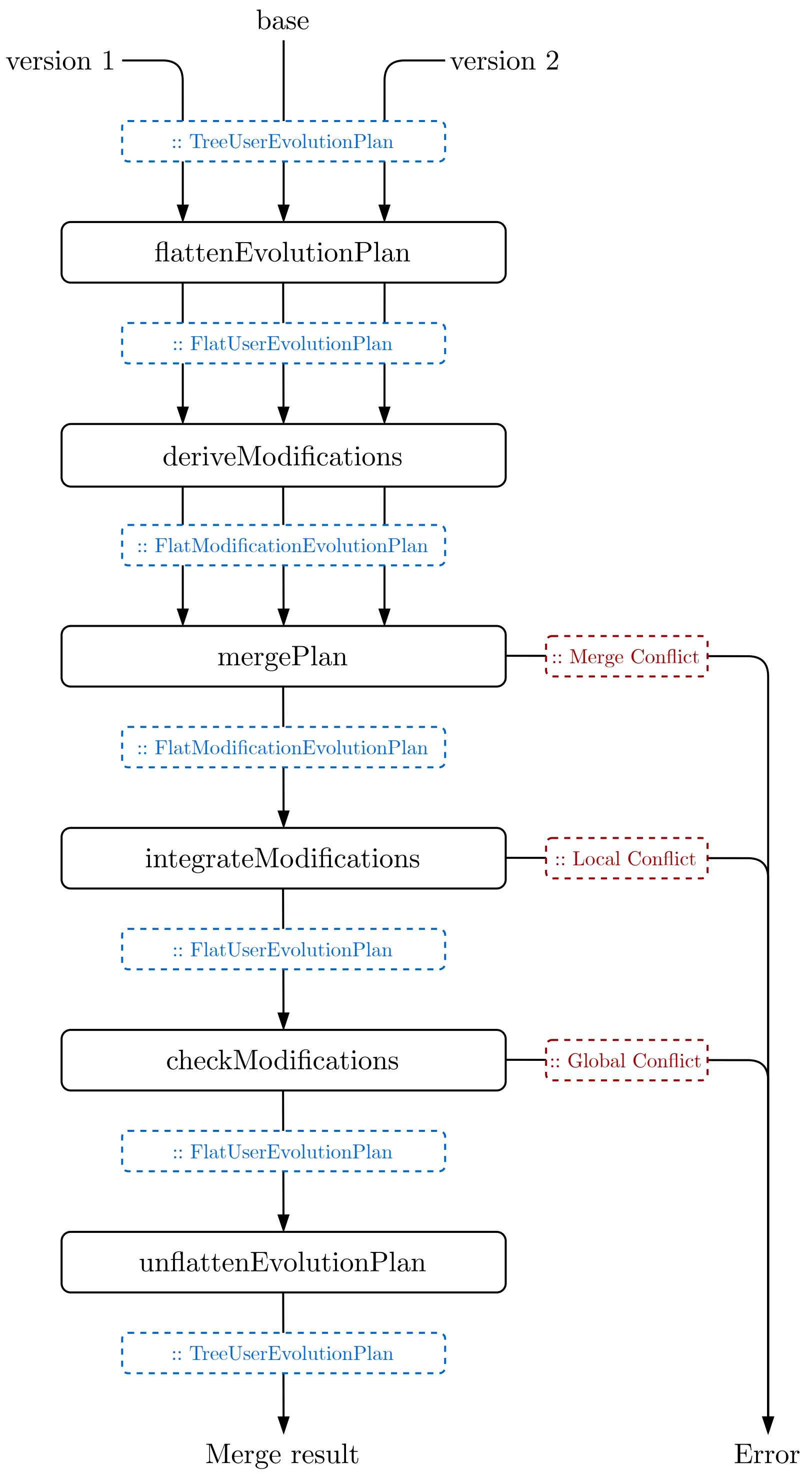


Figure 4.1: Outline of the three-way merge algorithm

The first phase is transforming the three different evolution plans into representations that is more suitable for merging. This includes converting both the way feature models are represented as well as the way the entire evolution plan is represented. This phase includes the `flattenEvolutionPlan` and `deriveModifications`, which is described in further detail in 4.2 on the following page

After changing the way evolution plans are represented, the second phase of the algorithm will calculate the differences between the *base* evolution plan and both derived evolution plans, *version 1*, and *version 2*. This will let us know what were added, changed and removed in each of the derived evolution plans. This phase is part of the `mergePlan` function, which is described in further detail in 4.3 on the next page

The information from the previous phase will be used to create a single merged evolution plan. This evolution plan is simply just the *base* evolution plan integrated with all the changes from *version 1* and *version 2*. This phase is part of the `mergePlan` function, which is described in further detail in 4.4 on the following page

Now that a single merged evolution plan is provided, the last step is to ensure that the plan is following the structural and semantic requirements of an evolution plan. Merging all changes from both versions might yield various inconsistencies. This includes structural conflicts such as orphan features, entire subtrees forming cycles, removing non-empty features, etc. The last phase includes converting back to the original representation, as well as ensuring soundness while doing so. This phase is part of the `integrateModifications`, `checkModifications` and `unflattenEvolutionPlan` functions, which is explained further in 4.5 on the next page

4.1.4 Conflicts

During the different phases of the merge algorithm, different kind of conflicts or errors could occur. Depending on what part of the algorithm a conflict occurred, the conflicts might be either a *merge*, *local* or *global* conflict. At what phase each conflict could occur can also be seen in Figure 4.1 on the preceding page, but a short description of the different conflicts are described below.

Merge Conflicts occur because of conflicting operations on a single feature or group. This could happen if one version tries to remove a feature, while the other tries to change the type of a feature. This could also happen if there originally existed a modification in the *base* version, and one of the derived versions try to change the modification, while the other tries to remove the modification.

Local Conflicts occur when a modification is not possible to be applied because of the existence or non-existence of a feature or group. For example, if we try to add a feature with an id that already exist, or try to change the type of a group that does not exist.

Global Conflicts is the last kind of error that could occur. When all the modifications has been integrated into the evolution plan, each feature model is checked for certain structural or semantical errors. At this point, each change *local* to a feature or group is valid, so we check for potential errors that occur because of dependencies between the features and groups, *global* to the entire feature model. The structural errors is typically modifications that lead to anomalies in the tree structure. These violations of the structure could happen if you add features to parents that don't exist, remove groups that has children, or move features in such a way that cycles are formed. Other violations to the semantics are also checked. This could for example be violations of well-formedness, that could happen if we change the type of a feature to something incompatible with its group.

4.2 Converting To a Suitable Representation

4.2.1 Representing Feature Models

4.2.2 Representing Evolution Plans

4.3 Detecting the Changes Between Versions

4.4 Merging Intended Changes

4.5 Ensuring structural and semantic soundness of the merge result

Chapter 5

Example Merge – Vending Machine

write about the vending machine example. Write about how the entire example is done. from cli parsing, converting to right representation, merging, checking, converting back, writing to file. Then how the frontend visualization tool parses the result and displays the tree as an interactive thing.

sound vs unsound examples

Chapter 6

Conclusion and Future Work

konklusjon bør si hvordan det jeg har gjort addresserer forskningsspørsmålene forskningsspørsmålene kan være større en bidraget how to ensure sound plan and merge of sound plans

future work kan være flere ting å se på for å belyse forskningsspm

Bibliography

[1] Jim Buffenbarger. "Syntactic Software Merging". In: *Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Selected Papers*. Ed. by Jacky Estublier. Vol. 1005. Lecture Notes in Computer Science. Springer, 1995, pp. 153–172. DOI: `10.1007/3-540-60578-9\_14`. URL: `https://doi.org/10.1007/3-540-60578-9%5C_14`.

[2] James W. Hunt and Thomas G. Szymanski. "A Fast Algorithm for Computing Longest Subsequences". In: *Commun. ACM* 20.5 (1977), pp. 350–353. DOI: `10.1145/359581.359603`. URL: `https://doi.org/10.1145/359581.359603`.

[3] 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`.

[4] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. "Parallel changes in large-scale software development: an observational case study". In: *ACM Trans. Softw. Eng. Methodol.* 10.3 (2001), pp. 308–337. DOI: `10.1145/383876.383878`. URL: `https://doi.org/10.1145/383876.383878`.

[5] Walter F. Tichy. "Tools for Software Configuration Management". In: *Proceedings of the International Workshop on Software Version and Configuration Control, January 27-29, 1988, Grassau, Germany*. Ed. by Jürgen F. H. Winkler. Vol. 30. Berichte des German Chapter of the ACM. Teubner, 1988, pp. 1–20.