Three-Way Semantic 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 Semantic Merge for Feature Model Evolution Plans

Eirik Halvard Sæther

Acknowledgements

[[TODO: write]]

<div align="right">

Eirik Halvard Sæther
Oslo, 2021

</div>

Abstract

A software product line (SPL) models closely related software systems by capitalizing on the high similarity of the products by organizing them in common and variable parts. Software engineers explicitly encode similarities and differences of an SPL by defining it in terms of a feature model. In order to ensure successful long-term development, it is beneficial to not just capture the current software product line, but the planned evolution of the SPL as well.

Evolution planning of an SPL is often a dynamic, changing process, due to changes in product requirements. In addition, planning is typically a collaborative effort with multiple engineers working separately and independently of each other. To improve development, their individual contributions would need to be synchronized and unified. This can be a complex task, especially without proper synchronization tools.

In this thesis, we develop a merge tool for evolution plans. The essence of the tool is a three-way merge algorithm. Given two different versions of an evolution plan, together with the common evolution plan they are derived from, the merge algorithm attempts to merge all the different changes from both versions. If the evolution plans are unifiable, the algorithm succeeds and yields the merged result containing the changes from both versions. However, if the changes are conflicting by breaking the structure or semantics of evolution plans, the algorithm terminates and reports the reason for failure. The three-way merge algorithm will act as an essential component in a version control system, allowing several contributors to synchronize their individual versions into a unified evolution plan.

Contents

List of Figures

List of Tables

Part I

Introduction and Background

Chapter 1

Introduction

A *Software Product Line (SPL)* is a collection of closely-related *software products*. The different software products leverage the high similarity by explicitly encoding the common and variable parts [1], [2]. As a software development paradigm, SPLs allow the engineers to model the software's commonality and variability to facilitate large scale reuse.

The most common variability model, *feature models (FM)* capture the different aspects and visible characteristics of a system in terms of features [3]. Each feature in the feature model corresponds to a certain increment in program functionality. The particular *variant* of the software product line is defined by a unique combination of features [4]. The feature model organizes the features in a tree structure to capture the variability of the SPL.

For instance, we can model common and variable features of a product line of cars in a feature model. Cars usually have several similarities in software, such as some sort of infotainment system. Other features are only present in some car models, e.g. electrical battery indicator. Modeling the car SPL in a feature model would dictate what combination of features are valid. Modeling SPLs features and dependencies between them in a feature model dictates what variants of cars are allowed to assemble.

Since SPLs undergo continuous evolution, planning for the long-term evolution of a software product line is often crucial [5]. While long-term planning is beneficial for success, there is little support for handling the long-term evolution [6]. The evolution of the SPL is usually handled as an informal procedure relying on the intuition and experience of individual engineers. Without formal tools, long-term goals might not be addressed properly, increasing the risk of significantly increased development costs.

As the evolution of SPLs should start by modifying the feature model [7], we address the lack of tools for long-term evolution planning with *feature*

model evolution plans, or just *evolution plans*. An evolution plan will not only model the current feature model, but all intended future feature models as well. This allows engineers as well as non-technical stakeholders to have a concrete tool for planning the long-term development of the software product line. The evolution plan makes it possible to plan when certain features are introduced or removed, or simply changing the way the features are related to each other.

1.1   Motivation

The use of version control systems are ubiquitous and a big part of the daily life of an engineer. Git[1] and other version control systems are frequently used to synchronize efforts of multiple engineers. Planning the evolution of an SPL often involves multiple engineers changing and evolving the plan. Having several engineers working in parallel on the same evolution plan can be beneficial in handling the dynamic nature of evolution planning. Therefore, tools supporting evolution planning could benefit from synchronization techniques allowing collaborators to work independently. Since naïvely merging multiple versions of an evolution plan may yield inconsistencies and conflicts, we investigate merging strategies respecting the structure and semantics of evolution plans.

1.2   Objective

The objective of my thesis is to design and implement a three-way merge algorithm for evolution plans. To achieve an effective and accurate algorithm, good data structures and representations for evolution plans has to be chosen. The three-way merge algorithm will consider a base evolution plan, and two derived evolution plans. The algorithm will then try to merge the two derived plans, with the base model as a reference to what changes were made. If the changes in both versions are incompatible, an error would be produced indicating why the evolution plans were not unifiable. However, the merge is successful if the changes are compatible and the resulting evolution plan is *sound*, meaning that the structure and semantics are not violated. The algorithm should follow some specified heuristics to ensure a plan that follows the users intent as closely as possible, without creating an unnecessary amount of conflicts and warnings. The algorithm should also find a good balance between complexity and usability.

---

[1] `https://git-scm.com/`

1.3   About the Project

The work done in this thesis is part of a larger research project, aiming to address the lack of support for planning the long-term evolution of software product lines. The project, *Long-Term Evolution Planning for Highly Variable Software Systems (LTEP)*, is a collaboration between *Technische Universität Braunschweig (TUBS)* and *University of Oslo (UiO)*. The project aims to strengthen academic relations between the countries, perform cooperative research at a high academic level and qualify young academics in an international environment. The LTEP project started in 2019, and was funded by the *Norwegian Research Concil (NFR)* and the *German Academic Exchange Service (DAAD)*.

The LTEP project aims to devise methodology for long-term evolution planning in highly variable software systems. Creating SPLs and performing evolutionary changes are research areas that have received significant focus. However, the planning of evolution in these highly variable software systems is largely overlooked by research to this day. Within the LTEP project, we created a novel variability modeling notation which allows collaborative planning of SPL evolution. We are the first to identify, analyze and circumvent evolution paradoxes to the best of our knowledge.

During the course of the LTEP project, we published a paper, *Consistency-Preserving Evolution Planning on Feature Models* [8]. In the paper, we presented methods for creating and retroactively adapting a feature model evolution plan while preventing structural and logical inconsistencies in the model. The method was established by structural operational semantics and linear temporal logic, and implemented using rewriting logic and integrated with the DarwinSPL tool [2].

While the published work addresses some of the scientific aims of the LTEP project, some issues still remain. Concretely, one of the scientific aims is harmonizing partial evolution plans of multiple contributors, which my thesis will attempt to address.

1.4   Research Questions

The overall goal of this thesis is to provide tools aiding developers and engineers cooperate in planning the evolution of a software product line. To achieve this goal, this thesis will address the following research questions (RQ):

---

[2] https://gitlab.com/DarwinSPL/DarwinSPL

RQ1 *How do we design an algorithm for merging evolution plans?* When designing an algorithm, we need to explore what characteristics the algorithm needs in order to merge contributions effectively. This includes figuring out what data structures and representations are suited, what approach we need for merging, etc.

RQ2 *How do we create a merge tool for evolution plans that respects soundness?* When developing a merge tool, we optimally want to make sure that the structure and semantics of evolution plans are met. How do we create a tool to ensure this?

RQ3 *How can we make predictable and interpretable results of the merge tool?* As we want the merge tool to be used by humans, the output of the tool needs to be possible to understand. When the merging is successful, the output needs to make sense to the user. When a conflict is produced, the user should also be able to understand the error message and have ways of dealing with the conflict. The results also need to be predictable, meaning that the merger does not produce unwanted results that go unnoticed by the user.

1.5   Contributions

The main contribution of this thesis is a three-way merge algorithm for feature model evolution plans that ensures a result respecting structure and semantics upon a successful merge. In order to achieve this, we have created a new representation of evolution plans more suitable for merging. With such a representation of evolution plans, the algorithm will include alterations from both derived versions, combining them into a single evolution plan, which is then checked to ensure a sound, well-formed evolution plan.

The three-way merge algorithm is implemented in the strongly typed, functional programming language Haskell. The Haskell program is created as an *command-line interface (CLI)*, to handle reading and writing from a file, logging the output of the algorithm, checking for correct behavior, etc. The CLI allows a variety of different input formats for the evolution plans, as well as the option for generating some predefined examples, both sound and unsound. Since feature models and evolution plans often are very hard to read in textual format, a frontend visualization tool has also been created using a language similar to Haskell, namely Elm. The Elm application is a web application that can display the evolution plans as visual tree structures, and lets users explore what the

merge algorithm produces. Several examples and test cases have also been implemented, checking that the expected behavior of the program matches the actual output.

[[TODO: finne ut hva ida mener]] To summarize, the contributions include:

- Created a formal definition of evolution plans suitable for merging

- Constructed a three-way merge algorithm producing sound, well-formed evolution plans

- Implemented the algorithm as part of a command-line interface in Haskell

- Created a visualization tool for exploring the results of the merge in Elm

- Provided test cases and examples that result in both sound evolution plans and merge conflicts

1.6 Chapter Overview

Chapter 2 introduces necessary background in order to understand the contributions in the thesis. This includes background on SPLs, feature models and evolution planning, a section about software merging, as well as an introduction to the programming language used in this thesis.

Chapter 3 defines the problem and the characteristics of the solution. We define soundness for evolution plans as well as a high-level overview of the three-way merge tool.

Chapter 4 covers implementation and specification of the merge algorithm. This includes the formalization of data structures, the specification of the functions used in the algorithm as well as a running example for demonstrating the implementation.

Chapter 5 outlines the tools supporting the core algorithm. This includes a command-line interface for automated interaction with existing tools as well as a tool for visualizing and exploring the merger.

Chapter 6 shows the execution of the merge tool on examples from an SPL inspired by real-world examples. It includes examples resulting in both success and failure.

Chapter 7 concludes the work of the thesis. This includes a discussion of the research questions as well as future work.

1.7 Project Source Code

All the source code from the master thesis can be found on Github[3].

---

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

Chapter 2

Background

In this section, we detail background information to better understand the contents of the thesis. This includes three sections. The first section is about software product lines, feature models, and evolution plans. This is important because it is the software artifact we are attempting to create merge tools for. The second section is about version control systems. We discuss different types of merging tactics, including their respective drawbacks and benefits. Lastly, we introduce the programming language and theory necessary to better understand the implementation details in this thesis.

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 make 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.

Several such variants are often needed to be developed in parallel to address a wide range of customer requirements [3]. Simultaneous development of variants is often a complex task, affecting all phases of development. SPLs often emerge *ad-hoc* when companies see the need for a similar yet different product than the existing ones. In many of these cases, this is handled by cloning and modifying the software of an existing product, known as *clone-and-own* [9]. The issue with clone-and-own often emerges when changes are required to the common parts of the variants. Since the different versions are separated entirely, changes need to be done in both versions. This can be problematic, especially in SPLs with a large number of variants.

Software product line engineering is a discipline for efficiently developing such families of software systems. Instead of maintaining potentially hundreds of different software variants, these engineering methods have ways of capitalizing on the similarities and differences between each variant. Such *variability realization mechanisms* are used to encode the variability in the implementation explicitly. Instead of maintaining one code base per variant, all variants are instead expressed in the same code base. There are two common ways of doing this, the *compositional approach* and the *annotative approach* [10]. The compositional approach models features in distinct components, allowing variants to be generated by *composing* components together. On the other hand, the annotative approach relies on annotating the source code similarly to how C-style preprocessors use `#ifdef` and `#endif` annotations. Depending on the software variant needed, the preprocessor uses the annotations to generate the correct software variant.

2.1.1 Feature Models

While tools for implementing software product lines are very important, planning and specifying the different variants of an SPL are also beneficial. Variability models play a central role in this planning and are also used as a communications tool. We will detail the most common variability model, the *feature model*, as it is an important software artifact this thesis relies on.

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 represent a concrete aspect in an SPL. Features can be mandatory or optional and will contain zero or more groups. Each group has a set of features and the type of the group dictates which features in the group can be selected. For example, in an *and* group, all the mandatory features have to be chosen, while *alternative* groups have to select exactly one feature. Normally, feature models also contain cross-tree constraints. However, since these are not in the scope of the thesis, we will not detail them here.

A visual representation of a feature model can be seen in Figure 2.1 on the following page. The small dot above `Infotainment System` indicates that the feature is mandatory, whereas 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 `Alternative` 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).

Note that the visualization in Figure 2.1 on the next page is one of the

Figure 2.1: Car Example - Feature Model

common representations of feature models. However, we will through the rest of this thesis use a slightly different notation for reasons we will explain in Section 3.4.3 on page 29.

2.1.2 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. Developing these kinds of systems typically involves many engineers, managers, or other stakeholders. Managing when certain changes, additions, or deprecations are implemented can be complex and confusing without suitable tools. Changing the SPL may also potentially influence many configurations, which might conflict with stakeholders' requirements. Thus, it is paramount to thoroughly plan SPL evolution in advance, e.g., to perform analyzes and to have enough time for implementing new or adapted features.

Several approaches and tools for planning the evolution of an SPL exist to this day. DeltaEcore is a tool suite relying on *Hyper-Feature Models (HFMs)* and *evolution delta modules* for the integrated management of variability in space and time in SPLs [11]. DarwinSPL is another tool suite for modeling evolving context-sensitive SPLs by extending DeltaEcore's HFMs [12]. EvoFM supports long-term feature-oriented planning and analysis of evolution plans [6]. In addition, other tools also include Feature-Driven Versioning [13], FORCE [14] and SuperMod [15].

In this thesis, we will focus our efforts on *Evolution Plans (EPs)*, also called *Feature Model Evolution Plans* [8]. EPs 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 at a point in time, which could have varying validity, from a week from now to

a year or more. 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 *time points*, where each time point is a set of operations to perform on the previous feature model to derive the next. The operations will either change, add, or delete features or groups in the feature model.

```
                         ┌──────────┐
                         │   Car    │
                         └──────────┘
                              │
                              ●
                    ┌────────────────────┐
                    │ Infotainment System │
                    └────────────────────┘
                              │
                              ○
                       ┌──────────┐
                       │ Bluetooth │
                       └──────────┘
```
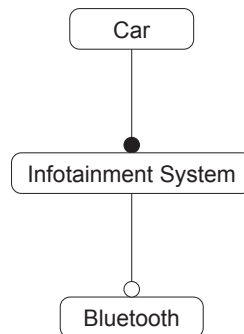
At time 1:
_____
add an Alternative group to Infotainment System.
add feature Android Auto to the Infotainment System Alternative group
add feature Car Play to the Infotainment System Alternative group

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: Car 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 times 1 and 2. Performing all the operations results in a feature model that is identical to the one in Figure 2.1 on the preceding page.

2.2   Version Control Systems

Software configuration mechanisms is the discipline of managing the evolution of large and complex software systems [16]. *Version control mechanisms* are used to deal with the evolution of software products. These mechanisms include methods of dealing with 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 [17] categorizes and describes different aspect of version control systems and software merging techniques. Two-way and three-way merging differentiate between how many versions of the artifact are comparing. Different representations of the merge artifact can be categorized in textual, syntactic, semantic, or structural merging. State-based merge techniques use delta algorithms to compute differences between revisions while change-based techniques keep track of the exact operations that were performed between the revisions.

2.2.1 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 merger 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 2 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")`.

2.2.2 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 compute the difference between files by comparing equality over the lines. This has several implications to how it calculates

changes between versions. Adding a single space after a line result in a deletion of the old line and addition of the same line with a single-spaced appended to it. Similar problems also occur when we change the indentation or formatting in the files. This coarse granularity will in many cases often lead to unnecessary and confusing conflicts.

To exemplify this, we will consider a simple Python example to showcase how textual, line-based merging can problematic. Since calculating the difference between a base version and a derived version is an important step in merging, we will use the Unix' *diff*-tool [18] to exemplify the shortcomings of the standard textual merging tools. First, we consider the following Python function.

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

With this python program as a starting point, we change the program and end up with the following, derived version.

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

The second version simply wrapped the content of the function in an if-statement that checks for input sanity. Running this example with the diff-tool, we get the following results.

```
<    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)
```

The resulting differences between the two versions are confusing and inaccurate. Conceptually, the difference is that the second version

wrapped the block in an 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.

As discussed, text-based merge techniques often provide inferior results, however, they have several advantages in terms of efficiency and generality. 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 [19]. 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 variable, while another version introduces new lines referencing the old variable.

2.2.3   Syntactic Merging

Syntactic merging [20] 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. For example, representing the Python program from Section 2.2.2 on page 12 as a parse tree or abstract syntax tree, we can represent the changes more accurately.

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.

2.2.4   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 constraints can be seen in 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 take 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.

2.3 Haskell and Algebraic Data Types

In this thesis, we will rely on Haskell and algebraic data types in order to specify and implement the three-way merge algorithm. Therefore, we give a short introduction to some of Haskell's most important aspects. Haskell is a purely functional, compiled, strongly typed programming language. While we will not go into full detail on the implications of these traits, we will discuss some of them.

Sum Types

The strongly typed nature of Haskell allows for precise and accurate formal definitions of our data types. Haskell's data types are rooted in algebraic data types[1]. One of its most important aspects is *sum types*, which allows us to define types that can have one or more variants. As an example, we show how a polymorphic binary tree is encoded in Haskell.

```
data Tree a
  = Empty
  | Node a (Tree a) (Tree a)
```

---

[1] https://en.wikipedia.org/wiki/Algebraic_data_type

Since the tree is polymorphic, we can choose what type the elements in the tree have. A simple tree of `Int` with three nodes can be encoded in the following way.

```
exampleTree :: Tree Int
exampleTree =
  Node 1
    (Node 2 Empty Empty)
    (Node 3 Empty Empty)
```

The benefits of the type system become clear when we define functions on the data types. When defining functions on sum types, we can *pattern match* on the data type. When we are pattern matching, Haskell's type system forces us to consider every case, yielding a compilation warning if we don't. To exemplify this, we consider a function on the newly defined `Tree` type, calculating the number of nodes in a tree.

```
numNodes :: Tree a -> Int
numNodes Empty = 0
numNodes (Node i left right) =
  1 + numNodes left + numNodes right
```

This function is also polymorphic, noted by the type signature `numNodes :: Tree a -> Int`. This means that if we run our tree of integers on the function, using `numNodes exampleTree`, we get 3 as the result.

Record Types

While sum types are very powerful and useful, Haskell also allows for *record types*, which lets us define a structure with named fields. We provide a simple bank account example to showcase records.

```
data BankAccount = BankAccount
  { name :: String
  , amount :: Int
  }
```

This definition allows us to create a `BankAccount` in the following way: `BankAccount "account" 100`.

Monads and Advanced Type Classes

Being a *pure* functional language, Haskell forbids mutable data and state changes by relying on mathematical functions. This means that when a function is called, Haskell guarantees that no side effect other than what is specified by the type system takes place. If a function takes a `String`

and returns an `Int`, then there is no mutation of variables, no printing, no fetching of remote data. This does not mean that side effects are not possible, they just have to be expressed explicitly through the type system. If a function returns an `Int`, but would like to print something in addition, the return type would be `IO Int`. If a function could potentially fail, a `Maybe Int` or `Either ErrorType Int` could be the return type.

The purity of the language requires us to be very explicit and precise in our function definitions. This could potentially cause a lot of boilerplate which impure languages would circumvent. However, as we will see, Haskell uses *monads* and other *type classes* to let us avoid much of the boilerplate necessary to do these tasks. We will detail a common task for monads, which is error handling. One way of doing error-handling is using the `Maybe` data type, which is defined in the following way.

```haskell
data Maybe a = Nothing | Just a
```

Since the `Maybe` data type is polymorphic, it works for all kinds of types. As an example, we will show how we can create a function that can fail by specifying that the return type is `Maybe Int`. We will use standard integer division as our example, which fails if we divide by zero.

```haskell
safeDiv :: Int -> Int -> Maybe Int
safeDiv a b =
  if b == 0
    then Nothing
    else Just (a `div` b)
```

Dividing 10 by 2 can be done using `safeDiv 10 2`, which will yield `Just 5`. However, if we divide by zero using `safeDiv 10 0`, we will get `Nothing`. Our function is now safe, but it comes with the drawback of having to explicitly deal with errors. Oftentimes, we want the errors to be propagated automatically, much like how standard exceptions work. Luckily, the `Maybe` data type implements a lot of type classes, such as `Monad`, `Applicative` and `Functor`. This allows us to avoid a lot of boilerplate that comes with composing programs that use `Maybe` as error handling.

Haskell also comes with built-in syntactic abstractions like `do`-notation that helps us work with data types implementing the `Monad` class. In the case for `Maybe`, the monad implementation makes sure that errors are propagated immediately. To exemplify this, we will use do-notation to automatically compose several calls to the `safeDiv` function. The do-notation will then handle unwrapping the `safeDiv` results, sparing us of boilerplate.

```haskell
doSomeDividing :: Maybe Int
doSomeDividing = do
```

```
res1 <- safeDiv 10 5
res2 <- safeDiv 10 2
res3 <- safeDiv 14 7
return $ res1 + res2 + res3
```

The result of `doSomeDividing` is `Just 9`. However, if one of the calls to `safeDiv` failed, we would get `Nothing` as a result instead.

Lenses for Nested Data Structures

Another common issue with pure languages is that updating structures can be challenging. Since mutation is not allowed, one cannot simply change a field in a structure. Instead, a copy of the structure would have to be made, only with the single field changed. When handling deeply nested data structures, this can be quite cumbersome. One solution to dealing with this issue, is using *functional references*, also called *lenses*.

A lens is a value that represents a certain field, often called a *focus*, in a structure. This could be a certain position in a list or a field in a record. The same lens can be used both for viewing and updating structures. Lenses themselves can also be composed with other lenses, which creates lenses that handle nested structures.

Two common lenses used in the *lens*[2] library is the `_1` and `_2` lenses. These two lenses focus on the first and second part of a tuple respectively. To create a lens that focus on nested tuples, the lenses can be composed like this: `_1._2`. With our newly created lens, we can perform both view and update operations on a nested tuple, `((1,2), "hello")`. Using `view (_1._2) ((1,2), "hello")`, we will get `1`, and using `set (_1._2) 100 ((1,2), "hello")`, we will get `((100,2), "hello")`.

---

[2]`https://hackage.haskell.org/package/lens`

Part II

A Soundness-Preserving Merger for Evolution Plans

Chapter 3

Developing a Sound Three-Way Merge of Evolution Plans

In this chapter, we define the general characteristics of our tool for merging evolution plans. We define precisely what we are trying to merge, and how we plan to do it. We discuss the context of the tool, as well as how it could be integrated with other tooling. We also define the outline of the merge algorithm at a high level, defining all the different steps involved in the algorithm. Precise formal semantics of evolution plans are also presented, due to their importance in creating a merger producing sound plans.

3.1   Problem Definition

Evolution planning is a long-term endeavour, often requiring planning years ahead when dealing with large scale SPLs. Software product lines are subject to frequent changes, which require replanning the evolution plan. An evolution plan consists of several feature models coupled with a time point, and making changes to intermediate feature models will affect the subsequent feature models. This is prone to errors, which yields the need for tools for ensuring that a plan is correct after applying changes to an evolution plan.

Evolution planning was invented to aid the development of long term software product lines. Development of these evolution plans often requires multiple engineers, often working in multiple teams. For this reason, we develop tools specifically for synchronizing the replanning efforts of multiple engineers. However, even though the individual evolution plans are conforming to the formal semantics of evolution plans, harmonizing their contributions might yield results that are not sound.

These issues might simply be diverging changes to the same thing, or more complicated violations of the semantics.

The core contribution of this thesis is a three-way merge algorithm. The algorithm takes three evolution plans as input, then output either an error or the resulting merged evolution plan. The three evolution plans consist of a base evolution plan, as well as two derived evolution plans, version 1 and version 2. The algorithm merges version 1 and version 2, using the base version to create better results. The results are merged in such a way that the result is guaranteed to follow the structure and semantics of evolution plans.

Three-Way Even though we are just trying to synchronize the efforts of two collaborators at a time, we are using three evolution plans to do so. The two evolution plans was not just created from scratch, but rather derived from a common, *base* evolution plan. As discussed in Section 2.2.1 on page 12, we can leverage the common evolution plan to more accurately derive what changes each version made.

Syntactic, Semantic Merger In contrast to the most common merging techniques, we will not opt for a textual merging technique. These techniques are often very general, and does not consider the merge artifact at hand. Since we know the structure and semantics of the merge artifact, we will design the merge algorithm as a *syntactic*, *semantic* merger. This simply means that the merger will take the structure and semantics of evolution plans into account when merging. The benefits of this approach is discussed in greater detail in Section 2.2.3 on page 14 and 2.2.4 on page 14.

Soundness Assumption The inputs of the algorithm, the three evolution plans, are assumed to be sound. The merge tool is designed with one focus, which is to synchronize evolution plans. This means that when the developers are at a stage where they want to merge their efforts, they have already ensured the correctness of their individual plans. This means that we can leverage the soundness of the input in the algorithms design.

3.2   Context of The Merger

In this section we will explain how the three-way merger will fit into the general work flow of engineers actually working on the evolution plans.

The merge tool is designed to be integrated as part of the evolution planning tool DarwinSPL[1]. In the DarwinSPL application, users are presented with a graphical interface for creating and modifying evolution plans. A screenshot of the application in use can be seen in Figure 3.1.

Figure 3.1: DarwinSPL Screenshot

As the merger is designed with the DarwinSPL application in mind, the merge tool is not intended to be interacted with directly. The tool we have created can act as backend for DarwinSPL, where the merger will have to be integrated as part of a bigger version control system inside the application.

A potential version control system can be tightly integrated with the editor as a wrapper layer between the editor and the three-way merge backend. Creating such a system can let users commit and push changes of the evolution plan. When multiple contributors are to harmonize their diverging versions, the diverging versions as well as the base plan they were derived from could be retrieved from the version control system and fed to the three-way merger. If the merge were successful, the engineer could check the output, and make small changes if necessary. If the merge resulted in a conflict, the conflict would be reported to the user, so that changes could be made in order to achieve a sound result.

---

[1]https://gitlab.com/DarwinSPL/DarwinSPL

Our three-way merge backend is implemented in Haskell, which is different from the DarwinSPL tool implemented in the Java language. For this reason, one could not simply integrate the three-way merge algorithm directly. To tackle this problem, we have defined a command-line interface facilitating the integration of the tools. The command-line interface includes serialization and deserialization of the input and output data structures, as well as several options and modes to control the way the merger behaves. This command-line interface is defined and discussed in more detail in Section 5.1 on page 91.

3.3   Algorithm Overview

In this section we describe the general outline of the merge algorithm. This includes the different phases involved, the potential conflicts that could occur as well as the different evolution plan representations we will encounter in the algorithm. Exactly how all the different parts interact with each other can be seen in an outline of the algorithm in Figure 3.2 on the following page.

3.3.1   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 3.2.

Converting Individual Plans   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 Section 4.2 on page 35.

Detecting Changes   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 page 52

Figure 3.2: Outline of the three-way merge algorithm

Unifying Changes   The information from the previous phase is used to create a single merged evolution plan. This evolution plan is simply 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 page 62

Ensuring Soundness   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 page 67

3.3.2   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 3.2 on the previous 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 apply 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   This is the last kind of error that could occur. When all the modifications have 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 are typically modifications that lead to anomalies in the tree structure. These violations of the structure could happen if you add features to parents that do not exist, remove groups that have 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 type.

3.3.3 Evolution Plan Representations

As seen in the merge algorihm outline in Figure 3.2, each step of the algorithm produces a certain representation for the evolution plans. This can be seen with the blue boxes between the phases, where we have the three representations `TreeUserEvolutionPlan`, `FlatUserEvolution-Plan` and `FlatModificationEvolutionPlan`. We will go into further detail about the different representations and their purpose later, but we give a brief overview below.

TreeUserEvolutionPlan   This is the representation that is closest to what the actual user would see and interact with. For this reason, we will use this as the main input type of the algorithm. In this representation, the evolution plan is modeled as an ordered list of feature models, where each feature model is modeled as a recursive tree structure. However, this representation of evolution plans are not well suited for detecting changes between versions as well as merging and unifying several plans.

FlatUserEvolutionPlan   In order to transform the input representation to a better suited representation for merging, we split the process up in two phases as described in Section 3.3.1 on page 23. This representation will be the intermediate representation in this process. The representation is similar to the `TreeUserEvolutionPlan` modeling the evolution plan as a list of feature models. The difference lies in the representation for feature models. Instead of a tree structure, we will model the feature model as a map of features and a map of groups indexed by their IDs.

FlatModificationEvolutionPlan   This is the final, merge-ready representation for evolution plans. In this representation, we keep the flat structure

of feature models from `FlatUserEvolutionPlan`. However, instead of having a list of feature models, we only have the initial feature model. Each subsequent time point will instead have a set of modifications necessary to transform the previous feature model into the next. Using this structure makes the process of the three-way merge significantly easier.

3.4 Formal Definition of Evolution Plans

As stated in **RQ2** in Section 1.4 on page 4, we want to design the merger in a way that guarantees a sound merge result. In order to do so, we need to have a clear, formal definition of the software artifact at hand, namely evolution plans.

The work we did in [8] defined the theoretical foundation of a sound, well-formed evolution plan, which we used to create a framework for ensuring soundness. In our merge tool, we are merging evolution plans which are assumed to be *sound*. This means we can leverage the properties of sound evolution plans defined in the paper. Since we also want to create a sound evolution plan, we will present the details necessary in order to understand the formal semantics of evolution plans.

3.4.1 Feature Models

Since evolution plans consist of feature models, we start by defining feature models formally. *Feature models* are a recursive tree structure of *features* and *groups*. Each feature model begins with a *root feature*, which has an arbitrary number of child groups. Each group has in turn an arbitrary number of features, etc. Each feature and group has also a unique ID. Each feature consists of an ID, name and a type. Each features type can either be *mandatory* or *optional*. Each group has an ID and a type, where the type of a group can either be *and*, *or* or *xor/alternative*.

Well-Formedness Requirements   In addition to the structure of a feature model, we also have a list of additional constraints restricting what is considered to be a sound feature model. These semantic rules are modeled as well-formedness requirements, and are listed below.

WF1 A feature model has exactly one root feature.

WF2 The root feature must be *mandatory*.

WF3 Each feature has exactly one unique name, variation type and (potentially empty) collection of subgroups.

WF4 Features are organized in groups that have exactly one variation type.

WF5 Each feature, except for the root feature, must be part of exactly one group.

WF6 Each group must have exactly one parent feature.

WF7 Groups with types *alternative* or *or* must not contain *mandatory* features.

Note that in the paper [8], there is an additonal well-formedness requirement stating that groups of type *alternative* or *or* must contain at least two child features. However, this requirement is not considered in this thesis.

3.4.2 Evolution Plans

In its most basic form, evolution plans are simply just a list of feature models, where each feature model is associated with a time point. This representation does not require any additional restraints other than the fact that the time points will have to be ordered.

Evolution planning often requires revisiting intermediate feature models, not just adding new feature models at the end of the evolution plan. By *replanning* the evolution plan and changing these intermediate feature models, the changes will be propagated in subsequent time points (i.e. adding a feature at a time point will result in the feature existing in subsequent time points).

In order to facilitate for replanning, we defined the semantics for what is considered a valid change on a feature model. A change was captured formally as an *operation*, which described how the feature model would be changed, and under what conditions. The operations include addition and deletion of features and groups, as well as modifications to their fields, such as names or types.

In the work of this thesis, I will build upon the semantics of the operations defined in [8]. However, we cannot use the operations directly, since due to several issues we will detail in Section 4.2.2 on page 39. Instead, we develop new representations which is better suited for merging evolution plans.

3.4.3 Visual Representation

As we want to visualize examples of feature models and evolution plans, we make *feature model diagrams* to achieve this. The visualization captures the most important aspects of a feature model in a visual tree structure.

The notation presented here is a bit different than what we have seen up until this point. The main difference is concerned with how groups are represented. In the common feature diagram notation, groups are often implicit relations between features. In cases with groups without features and *and* groups, the group is not visible in the notation, which can make results from the merge tool harder to understand. The notation presented here encodes groups similarly to features, which is intended to help the reader understand how the merger operates.

In the diagram, each feature is a node of rectangular shape with the name of the feature at the center of the node. Each feature has either a black or white circle above it, indicating whether the feature is mandatory or optional. The exception is the root feature, since the root is always mandatory. Groups are modeled as a circle with a symbol inside. The symbol inside represents the type of the group, where *and* groups use the $\wedge$ symbol, *or* groups use $\vee$, and *alternative* groups use the $\oplus$ symbol.
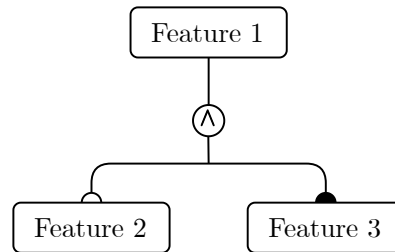
Figure 3.3: Visual representation of an example feature model

We present an example, seen in Figure 3.3, to demonstrate this. In this example, we have three features and one group. The three features is named *Feature 1*, *Feature 2* and *Feature 3*. Every feature is mandatory, except feature 2, which is optional. Since the group has the $\wedge$ symbol, it is an *and* group.

Chapter 4

Specification and Implementation of the Three-Way Merge

In this chapter, we present the three-way merge algorithm. The algorithm is formalized in the functional language Haskell. By leveraging Haskell's type system, we model different evolution plan representations, types of conflict and other intermediate representations essential to the algorithm. In addition to defining the types, we define the different parts and phases of the algorithm in terms of Haskell functions. We also demonstrate the transformations by constructing a simple example that we use in the entire chapter. Lastly, we discuss the time complexity of the algorithm.

4.1   A Normal Form for Evolution Plans

Before diving into the inner workings of the three-way merge algorithm, we define the input of the algorithm. Since the input of the algorithm is three evolution plans, we need to formalize evolution plans in Haskell's type system. When choosing a representation for the input, we model it in a way that is well aligned to the visual representation the users are presented with in the evolution plan editor. We call this representation the *normal form* of evolution plans, since it is closest representation to what the user is presented with.

In this representation, `TreeUserEvolutionPlan`, the evolution plan is represented as a list of feature models together with a time point. Each feature model is represented as a mutually recursive tree structure of features and groups.

The exact representation of `TreeUserEvolutionPlan` are defined formally in the Haskell code. Haskell's powerful type system with records

and algebraic data types allows for a precise formalization. When
presented here in the thesis, the types are somewhat simplified, leaving
out unnecessary noise, such as automatically derived instances for JSON
serialization, equality checking, etc.

4.1.1 Formalization of the Evolution Plan Normal Form

First, we look at the tree representation for feature models, formalized as
`TreeFeatureModel`.

```
data TreeFeatureModel = TreeFeatureModel
  { rootFeature :: TreeFeature
  }

data TreeFeature = TreeFeature
  { id :: FeatureId
  , featureType :: FeatureType
  , name :: String
  , groups :: Set TreeGroup
  }

data TreeGroup = TreeGroup
  { id :: GroupId
  , groupType :: GroupType
  , features :: Set TreeFeature
  }

data FeatureType
  = Optional
  | Mandatory

data GroupType
  = And
  | Or
  | Alternative

type FeatureId = String

type GroupId = String
```

There are a few important things to note in this representation. Each
feature's and groups's ID is unique across the entire feature model, which
we leverage when merging. Each feature and group can have an arbitrary
number of children. The children are organized in a Set, not a List,

31

noting that the ordering is irrelevant. In order for the feature model to be sound, some combinations of parent group types and child feature types are prohibited. If a group is of type `Alternative` or `Or`, every child feature has to be of type `Optional`.

Now that we have a suitable definition of feature models, we define evolution plans. In this representation, we want the evolution plans to mirror what the user is presented with, so we define the evolution plan as a list of feature models. In later stages of the merge algorithm, we will reuse the polymorphic `UserEvolutionPlan` definition of evolution plans, only with a different definition of feature models. This allows us to leverage Haskell's polymorphic type system, which spares us from redefining it.

```haskell
data UserEvolutionPlan featureModel = UserEvolutionPlan
  { timePoints :: [TimePoint featureModel]
  }

type Time = Int

data TimePoint featureModel = TimePoint
  { time :: Time
  , featureModel :: featureModel
  }
```

Now that a generalized representation for evolution plans are defined, using a representation with a list of feature models, we can instantiate the polymorphic evolution plan to create our normal form, `TreeUser-EvolutionPlan`. This is relatively straight forward, the only thing we have to do is to replace the `featureModel` argument with our concrete feature model, namely `TreeFeatureModel`.

```haskell
type TreeUserEvolutionPlan = UserEvolutionPlan TreeFeatureModel
```

4.1.2 Constructing a Simple Evolution Plan Example

Now that we have defined the normal form for evolution plans formally, we will give a concrete example and how it is represented in this formalization. This example will act as a running example through the rest of the chapter. In Section 4.2 on page 35, we will show how this example is transformed in the various evolution plan representations. In Section 4.3 on page 52, we extend this example to not just involve a single evolution plan, but three evolution plans. The example presented here will then act as the base evolution plan, while we construct two derived versions to showcase the merge process. The expanded example will continue through the rest of the sections in this chapter.

The simple example we will showcase is a small example containing three time points. The initial feature model at time 0 is simply just the root feature. The next time point at time 1 adds a new group and two features belonging to this group. The last time point removes one of the features and alters the name of the root and type of the group. A visualization of this simple evolution plan can be seen in Figure 4.1. Below is the Haskell code necessary for encoding this example.
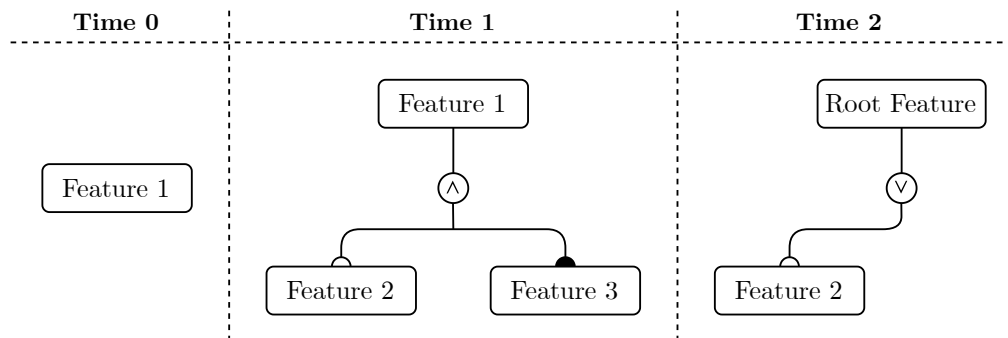
Figure 4.1: A Simple Evolution Plan Example

```haskell
simpleExample :: TreeUserEvolutionPlan
simpleExample =
  UserEvolutionPlan
    [ TimePoint 0 fm0
    , TimePoint 1 fm1
    , TimePoint 2 fm2
    ]
  where
    fm0 =
      TreeFeatureModel
        ( TreeFeature
            "rootFeature"
            Mandatory
            "Feature 1"
            []
        )
    fm1 =
      TreeFeatureModel
        ( TreeFeature
            "rootFeature"
            Mandatory
            "Feature 1"
            [ TreeGroup
```

```
                    "group"
                    And
                    [ TreeFeature
                        "feature2"
                        Optional
                        "Feature 2"
                        []
                    , TreeFeature
                        "feature3"
                        Mandatory
                        "Feature 3"
                        []
                    ]
                ]
            )
        fm2 =
          TreeFeatureModel
            ( TreeFeature
                "rootFeature"
                Mandatory
                "Root Feature"
                [ TreeGroup
                    "group"
                    Or
                    [ TreeFeature
                        "feature2"
                        Optional
                        "Feature 2"
                        []
                    ]
                ]
            )
```

Note that each feature and groups `Set` of children is constructed using list syntax. This is to make it a bit easier to read, not having too much visual clutter. This is possible using the language extension `OverloadedLists` [1], where we allow Haskell's type system to figure out that the list is actually a `Set` based on the context.

---

[1] `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/overloaded_lists.html`

4.2 Converting To a Suitable Representation

The evolution plan defined in Section 4.1.1 on page 31, `TreeUser-EvolutionPlan`, works well for capturing the essence of evolution plans. It represents evolution plans in terms of what the user sees and interacts with. However, in order to do an effective merge of evolution plans, we need to represent them a bit differently.

Evolution planning is not just done one time, but is a continuous process that requires changes according to new business requirements. This often requires changing intermediate time points in the evolution plan, not just the last one. In such events, the additions, deletions and modifications to features and groups have an effect not just on the time point at hand, but all subsequent time points in the evolution plan. When a user adds a new feature in the middle of the plan, the feature would appear on the specified time points as well as all the time points beyond. When calculating the changes made, we don't want to look at this like a feature was added in all the time points, but rather just the specified time. In order to capture the essence of the changes made to each evolution plan, we would need to find a representation capturing the actual changes between time points explicitly.

We will transform the evolution plan to the merge-ready representation, `FlatModificationEvolutionPlan`, in two steps. We will begin by briefly explaining the two steps in the paragraphs below, then in Section 4.2.1 on the next page and Section 4.2.3 on page 42 go into more detail on formal definitions, algorithms and examples.

A Flattened Structure For Feature Models To capture the explicit modifications between each time point in the evolution plan, we would need to figure out an approach to calculating the differences between two subsequent feature models. However, with our current tree-based feature model, `TreeFeatureModel`, this can be a bit cumbersome, requiring us to traverse the two trees simultaneously. In some cases this is not very straightforward, i.e. handling `Move`-operations that relocates entire subtrees. We will combat this by using a flat mapping structure for feature models, relying on node IDs for indexing. With this structure, calculating the differences between feature models becomes significantly easier.

Explicit Changes Between Subsequent Feature Models With a representation more suitable for calculating differences between feature models, we can transform the list of feature models to a merge-ready representation. This representation is modeled with an initial feature model and

a list of time points coupled with *modifications*. The modifications specify the exact changes between the previous feature model and the next. With this representation, knowing what changes each derived evolution plan versions has made is more explicit, allowing a more straight forward merge.

4.2.1 A Flattened Feature Model Structure

We define a new representation for feature models, `FlatFeatureModel`, in order to have a structure better suited for detecting and applying changes to the tree structure. This is achievable due to our features and groups having unique IDs. This allows for a simple mapping-structure, where we each feature and group can be looked up by its ID. The edges and relations in the tree are modeled as node-ID references instead of a recursive structure.

The new definition of feature models makes deriving and integrating modifications between two subsequent feature models easier. Since we have a mapping-based structure instead of a recursive tree structure, we do not need to recursively traverse both trees simultaneously in order to figure out what modifications has been made between the two trees. Cases where many subtrees are moved are especially challenging with a tree-based structure. The flat structure leverages the unique IDs in order to allow lookup based on ID. Changing a feature or group requires a lookup based on the ID, then changing the fields of the mapping entry. Removing or adding a node is similarly simple, requiring only adding or removing an entry in the mapping. Since the edges in the tree are modeled as references to the parent ID, we do not need to modify the parent node when removing a node. This makes moving entire subtrees straightforward, requiring changing only the parent field of the node being moved.

Formalizing the Flat Structure

We start by defining the new representation for feature models.

```
data FlatFeatureModel = FlatFeatureModel
  { rootId :: FeatureId
  , features :: Map FeatureId FlatFeature
  , groups :: Map GroupId FlatGroup
  }

data FlatFeature = FlatFeature
  { parentGroupId :: Maybe GroupId
```

```
  , featureType :: FeatureType
  , name :: String
  }

data FlatGroup = FlatGroup
  { parentFeatureId :: FeatureId
  , groupType :: GroupType
  }
```

The definitions of `FeatureType`, `FeatureId`, `GroupType` and `GroupId` are still the same as defined in Section 4.1.1 on page 31.

With our new, flat representation for feature models, we can use this to create a new user level representation, with our flat structure instead of the tree-based. Since we created a generalized data type, `UserEvolution-Plan`, we can instantiate it in the following way.

```
type FlatUserEvolutionPlan = UserEvolutionPlan FlatFeatureModel
```

Continuing the Simple Example

With this new representation for evolution plans, the example from Figure 4.1 on page 33 can be encoded in the following way:

```
simpleExampleFlat :: FlatUserEvolutionPlan
simpleExampleFlat =
  UserEvolutionPlan
    [ TimePoint 0 fm0
    , TimePoint 1 fm1
    , TimePoint 2 fm2
    ]
  where
    fm0 =
      FlatFeatureModel
        "rootFeature"
        [
          ( "rootFeature"
          , FlatFeature Nothing Mandatory "Feature 1"
          )
        ]
        []

    fm1 =
      FlatFeatureModel
        "rootFeature"
        [
```

37

```
                  ( "feature2"
                  , FlatFeature (Just "group") Optional "Feature 2"
                  )
                ,
                  ( "feature3"
                  , FlatFeature (Just "group") Mandatory "Feature 3"
                  )
                ,
                  ( "rootFeature"
                  , FlatFeature Nothing Mandatory "Feature 1"
                  )
                ]
                [("group", FlatGroup "rootFeature" And)]
        fm2 =
          FlatFeatureModel
            "rootFeature"
            [
              ( "feature2"
              , FlatFeature (Just "group") Optional "Feature 2"
              )
            ,
              ( "rootFeature"
              , FlatFeature Nothing Mandatory "Root Feature"
              )
            ]
            [("group", FlatGroup "rootFeature" Or)]
```

Each feature model in the evolution plan consists of a reference to the root feature ID, as well as two `Maps`. Using the `OverloadedLists` extension, each `Map` is noted by a list of tuples, where the first part of the tuple is the ID of the feature or group, and the second part consists of the fields of the feature or group.

Transformation from the Normal Form

The first step in the merge algorithm is using `flattenEvolutionPlan` to transform the tree-based evolution plan into the flattened structure. Note that in the code we call this `flattenSoundEvolutionPlan`, since we assume that the three plans in the algorithm are sound. In order to do this transformation, we need to go through every feature model in the plan and transform it using `flattenSoundFeatureModel`

```
flattenSoundEvolutionPlan ::
  TreeUserEvolutionPlan ->
  FlatUserEvolutionPlan
```

```haskell
flattenSoundEvolutionPlan =
  L.timePoints
    . traversed
    . L.featureModel
    %~ flattenSoundFeatureModel
```

Flattening a feature model is relatively straightforward, requiring traversing the tree structure, and creating a list of features and a list of groups as we traverse. In order to do so, we create a tuple of lists, where the left side contains a list of flat features and the right side a list of flat groups. When traversing the tree, we create a new flat feature or group, and concatinate it with the result from the recursive call with `<>` and `foldMap`.

```haskell
flattenSoundFeatureModel ::
  TreeFeatureModel ->
  FlatFeatureModel
flattenSoundFeatureModel fm =
  FlatFeatureModel
    (fm ^. L.rootFeature . L.id)
    (M.fromList features)
    (M.fromList groups)
  where
    (features, groups) =
      flattenFeature Nothing (fm ^. L.rootFeature)
    flattenFeature parent (TreeFeature id fType name gs) =
      ([(id, FlatFeature parent fType name)], [])
        <> foldMap (flattenGroup id) gs
    flattenGroup parent (TreeGroup id gType fs) =
      ([], [(id, FlatGroup parent gType)])
        <> foldMap (flattenFeature (Just id)) fs
```

4.2.2 Avoiding the Pitfalls of an Operation-based Representation

The next step in the merge algorithm is finding a representation for evolution plans where changes between time points are explicitly modeled. In doing so, we revisit the representation we created in the paper [8], as it is very similar to what we want to achieve. This operation-based representation expressed changes between feature models as a list of operations. However, this representation proposes a number of challenges, which we will discuss. In the light of these issues, we will in Section 4.2.3 on page 42 detail a more suitable solution for merging that tackles these issues.

In the operation-based semantic, we expressed changes between feature models as a list of operations. The operations covered the basic types of

modifications we could perform on features and groups, like `addFeature`, `removeGroup`, `moveFeature` and `renameFeature`. The effects and conditions of the operations are also formally defined in the paper. These operations are organized in a list, implying that the ordering is important. By applying the operations one by one to the current feature model, we get the next feature model.

The list of operations also reflected the way the user preformed the replanning, where the changes made had a specific ordering. However, when we design an approach to merging two evolution plans, the exact way each feature model in the evolution plan was derived is besides the point. The only thing the user is presented with in the evolution plan editor are a list of feature models associated with time points. Exactly how the each individual feature model was manipulated should not be relevant for the merge algorithm. For instance, if two features are added to a certain feature model, the order of which the features was added are not relevant.

We will detail some properties caused by the list-based operation semantic, and how they might be problematic for our purpose. The properties we will investigate are *non-dependent operations*, *dependent operations* and *operation shadowing*, which is discussed in the sections below.

Non-dependent Operations

In some cases, swapping the order of the operations in a time point has no effect on the resulting feature model. To exemplify this, take a simple feature model with two features and a group, organized in the following way: `Root` $\rightarrow$ `G1` $\rightarrow$ `F1`. In the following time point, there is scheduled a removal of feature `F1` as well as a name change for the root feature `Root` to `F0`. Applying both operations should yield the following feature model: `F0` $\rightarrow$ `G1`. As we can see, the order of when the operations are applied makes no difference. With the current list-based representation of a plan for a single time point, there are two ways of representing the changes. You could either schedule the removal of the `F1` feature first, or the name change of the root feature first. This example is visualized in Table 4.1.

This means that there are several ways to achieve the same evolution plan, which is a problem for our purpose. When users are merging what they perceive as the same evolution plan, we do not want to raise a conflict due to differences in the internal representation of evolution plans.

To counter these issues, one might naïvely find an standardized ordering of operations so one could sort the operations, or simply just store them as a set of operations instead of a list. However, as we will see, the order of
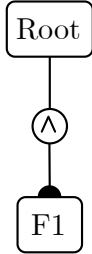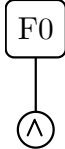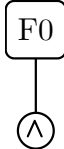
Initial Model	Operations	Original Order	Swapped Order
Root — ∧ — F1	Rename(Root, F0) Remove(F1)	F0 — ∧	F0 — ∧

Table 4.1: Non-Dependent Operations Example

the operations still matters, and changing the ordering can have varying effects, including both dependent and shadowed operations, which we will discuss further below.

Dependent Operations

In the previous example, switching the ordering of operations had no effect. In this example, we demonstrate an example where swapping the order has an effect. To showcase the problem, we start with the same feature model used previously, with two features and a group: `Root` $\rightarrow$ `G1` $\rightarrow$ `F1`. In the next planned time point, the changes to the feature model include the removal of both group `G1` and feature `F1`. However, there is only one legal ordering, namely removing the feature first, then removing the group. Applying the operation in this order yields a sound feature model; `Root`. Trying to remove the group first will as we defined sound evolution plans yield an inconsistency, because we are trying to remove a non-empty group. This example is also visualized in Table 4.2.
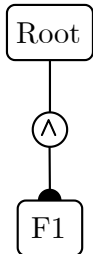
Initial Model	Operations	Original Order	Swapped Order
Root — ∧ — F1	Remove(F1) Remove(G1)	Root	Conflict: Removal of non-empty group

Table 4.2: Dependent Operations Example

Operations Shadowing

Another issue with the list-based approach is *operation shadowing*, which is the phenomenon where an operation is rendered useless because of another operation later in the list. This issue would also have to be considered, because rearranging the operations could lead to unwanted results. In addition to rearrangement resulting in unsound plans, moving a shadowed operation might result in a different, yet sound evolution plan.

We exemplify operation shadowing by considering the same initial feature model as before: `Root` → `G1` → `F1`. The list of operations for the next point in time include first changing the name of `F1` to `Feat 1`, then later to `Feature 1`. The operation shadowing occurs because of the second renaming makes the first renaming completely useless. The resulting feature model, `Root` → `G1` → `Feature 1`, is identical to the one we would have if we excluded the shadowed operation. However, if we were to swap the ordering of the rename operations, the resulting feature model would still be sound, but differ from the original ordering: `Root` → `G1` → `Feat 1`. This example is also visualized in Table 4.3.
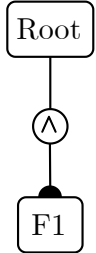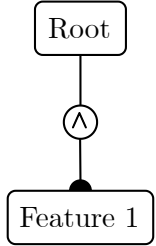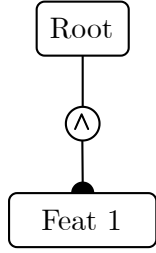
Initial Model	Operations	Original Order	Swapped Order
Root ∧ — F1	Rename(F1, Feat 1) Rename(F1, Feature 1)	Root ∧ — Feature 1	Root ∧ — Feat 1

Table 4.3: Shadowed Operations Example

4.2.3   A Merge-Ready Evolution Plan Representation

As discussed in Section 4.2.2, the list-based operation approach is problematic in order to achieve a desired merge result. We present the merge-ready representation for evolution plans. Similar to the operation-based representation, it will define a data type for deriving the next feature model from the previous. However, it will avoid the issues caused by an ordered list of operations.

We present the merge-ready evolution plan representation, `Flat-ModificationEvolutionPlan`, modeling the evolution plan as an initial

model along with an ordered list of time points associated with *modifications*. The modifications model the changes necessary to go from the previous feature model to the one in the specified time point. The modifications consist of two mappings, one for features and one for groups. Each mappings map IDs to a *modification*. By creating an indexed mapping structure, we limit a feature or group to have at most one modification at a time point. This could be either an addition, a removal, or a change to one or more fields.

The mapping structure also implies that there is no ordering to the feature and group modifications. Having no specific ordering to the modifications is challenging with the tree structured feature models. However, all these issues disappeared when we converted to the flat feature model structure, `FlatUserEvolutionPlan`. If we were to add a child before its parent, the tree structure makes this impossible. This is however made possible with the flexible flat structured feature models.

The new semantics for detecting changes between feature models has an important implication. The operation-based semantics allows verifying soundness after every operation application. However, since the mapping-based structure has no specific order of application, the verification would have to be postponed until every modification for a time point has been included. This is not relevant for now, but will be in further stages of the algorithm.

Formalizing Modification-Based Evolution Plans

We define the modification-based representation of evolution plans in this section. The Haskell representation of our `FlatModificationEvolution-Plan` can be seen below.

```haskell
data TransformationEvolutionPlan transformation featureModel =
  TransformationEvolutionPlan
    { initialTime :: Time
    , initialFM :: featureModel
    , plans :: [Plan transformation]
    }

data Plan transformation = Plan
  { timePoint :: Time
  , transformation :: transformation
  }

type ModificationEvolutionPlan featureModel =
  TransformationEvolutionPlan Modifications featureModel
```

```
type FlatModificationEvolutionPlan =
  ModificationEvolutionPlan FlatFeatureModel
```

Notice that the actual types are generalized in two ways. As with our user level representation, we have made the evolution plan polymorphic over the feature model. In addition, we have generalized the actual transformation type necessary to go from one feature model to the next. This is useful for the actual merge algorithm, which reuses the evolution plan with another transformation type. With the evolution plan structure in place and our newly defined `ModificationEvolutionPlan`, we can define `Modifications` in the following way:

```
data Modifications = Modifications
  { features :: Map FeatureId FeatureModification
  , groups :: Map GroupId GroupModification
  }

data FeatureModification
  = FeatureAdd GroupId FeatureType String
  | FeatureRemove
  | FeatureModification
      (Maybe FeatureParentModification)
      (Maybe FeatureTypeModification)
      (Maybe FeatureNameModification)

data FeatureParentModification
  = FeatureParentModification GroupId

data FeatureTypeModification
  = FeatureTypeModification FeatureType

data FeatureNameModification
  = FeatureNameModification String

data GroupModification
  = GroupAdd FeatureId GroupType
  | GroupRemove
  | GroupModification
      (Maybe GroupParentModification)
      (Maybe GroupTypeModification)

data GroupParentModification
  = GroupParentModification FeatureId

data GroupTypeModification
```

```
      = GroupTypeModification GroupType
```

Merge-Ready Representation of the Simple Example

We revisit the simple example introduced in Section 4.1.2 on page 32. With our final, merge-ready representation of evolution plans, we can encode the simple example in the defined data types. A visual representation of this encoding can also be seen in Figure 4.2 on the next page.

```
simpleExampleMod :: FlatModificationEvolutionPlan
simpleExampleMod =
  TransformationEvolutionPlan
    0
    initial
    [Plan 1 modifications1, Plan 2 modifications2]
  where
    initial =
      FlatFeatureModel
        "rootFeature"
        [
          ( "rootFeature"
          , FlatFeature Nothing Mandatory "Feature 1"
          )
        ]
        []
    modifications1 =
      Modifications
        [
          ( "feature2"
          , FeatureAdd "group" Optional "Feature 2"
          )
        ,
          ( "feature3"
          , FeatureAdd "group" Mandatory "Feature 3"
          )
        ]
        [("group", GroupAdd "rootFeature" And)]
    modifications2 =
      Modifications
        [ ("feature3", FeatureRemove)
        ,
          ( "rootFeature"
          , FeatureModification
              Nothing
```

```
        Nothing
        (Just (FeatureNameModification "Root Feature"))
    )
]
[
  ( "group"
  , GroupModification
      Nothing
      (Just (GroupTypeModification Or))
  )
]
```

Time 0	Time 1	Time 2

Feature modifications

"feature2": Add Feature:
    parent = "group"
    type = Optional
    name = "Feature 2"

"feature3": Add Feature:
    parent = "group"
    type = Mandatory
    name = "Feature 3"

Feature modifications

"feature3": Remove Feature

"rootFeature": Name Modification:
    new name = "Root Feature"

Feature 1

Group modifications

"group": Add Group:
    parent = "rootFeature"
    type = And

Group modifications

"group": Type Modification:
    new type = Or

Figure 4.2: A Simple Evolution Plan Example

Specifying the Final Representation Transformation

Up until this point, we have defined the data types involved and represented the simple example in the newly defined types for the modification-based evolution plan. We will now begin the process of specifying the algorithm performing this transformation.

The function uses the first time point in the evolution plan as the initial time point, and create a list of subsequent feature model pairs which is then passed to the `timePointsToPlan` function. The function will use the two subsequent feature models, previous and current, to calculate the differences between them.

As before, we assume soundness for our plans, which is why the actual function in the code is called `deriveSoundModifications` and not `derive-Modifications` as in Figure 3.2.

```
deriveSoundModifications ::
  FlatUserEvolutionPlan ->
  FlatModificationEvolutionPlan
deriveSoundModifications (UserEvolutionPlan timePoints) =
  case timePoints of
    [] -> error $ "evolution plan has to have "
               ++ "at least one time point!"
    ((TimePoint initialTime initialFM) : restTimePoints) ->
      TransformationEvolutionPlan
        initialTime
        initialFM
        (zipWith timePointsToPlan timePoints restTimePoints)


timePointsToPlan ::
  TimePoint FlatFeatureModel ->
  TimePoint FlatFeatureModel ->
  Plan Modifications
timePointsToPlan
  (TimePoint _ prevFM)
  (TimePoint currTime currFM) =
    Plan currTime $ diffFeatureModels prevFM currFM
```

The main part of the algorithm is actually detecting the changes between the two feature models and representing them in the `Modifications` data type. The function defined below, `diffFeatureModels`, handles this process. Since the flat representation of feature models includes two maps, one for features and one for groups, we calculate the modifications for both separately, using the functions `calculateFeatureModifications` and `calculateGroupModifications`.

```
diffFeatureModels ::
  FlatFeatureModel ->
  FlatFeatureModel ->
  Modifications
diffFeatureModels prevFM currFM =
  Modifications
    ( calculateFeatureModifications
        (prevFM ^. L.features)
        (currFM ^. L.features)
    )
    ( calculateGroupModifications
        (prevFM ^. L.groups)
```

```
      (currFM ^. L.groups)
  )
```

In order to calculate the differences between two `Maps`, we rely on the Haskell module `Data.Map.Merge` from the `containers` module[2]. This module provides ways of merging maps with keys of the same type. In our case, we have two mappings from `FeatureId` to `FlatFeature`, which we will attempt to merge into a mapping from `FeatureId` to `FeatureModification`. The merging is done using the `merge` function from `Data.Map.Merge`, which compares the keys, the feature IDs, in both maps. Based on the result of the comparison, one of three different merge tactics is used to produce the desired result. The merge tactics relies on the three functions that we pass to the `merge` function. Given a feature ID, the following three cases can appear:

- **The ID exists only in the previous feature model**: This will produce a `FeatureRemove` modification for the ID.

- **The ID exists only in the next feature model**: Since the feature did not exist in the previous feature model, but appeared in this, we will get a `FeatureAdd` modification for the ID.

- **The ID exists in both the previous and next feature models**: If the two features are identical, no modification should be generated. However, if any of the fields has changed, a `FeatureModification` would be generated. The modification will include all the fields that were changed, and skip the ones that remained unchanged.

```
calculateFeatureModifications ::
  Map FeatureId FlatFeature ->
  Map FeatureId FlatFeature ->
  Map FeatureId FeatureModification
calculateFeatureModifications =
  Merge.merge
    (Merge.mapMissing (const inPrev))
    (Merge.mapMissing (const inNew))
    (Merge.zipWithMaybeMatched (const inBoth))
  where
    inPrev _ = FeatureRemove
    inNew (FlatFeature mParent featureType name) =
      case mParent of
        Nothing -> error "cannot add a new root"
        Just parent -> FeatureAdd parent featureType name
    inBoth prev new =
      let FlatFeature prevParent prevType prevName = prev
          FlatFeature newParent newType newName = new
```

---

[2]https://hackage.haskell.org/package/containers-0.6.2.1

```haskell
        in if prev == new
             then Nothing
             else Just $
               FeatureModification
                 ( case (prevParent, newParent) of
                     (Just prev, Just new)
                       | prev /= new ->
                         Just (FeatureParentModification new)
                     -- NOTE: since the root is assumed to
                     ↪  never change, we only record changes
                     ↪  of non-root features
                     _ -> Nothing
                 )
                 ( if prevType == newType
                     then Nothing
                     else Just (FeatureTypeModification newType)
                 )
                 ( if prevName == newName
                     then Nothing
                     else Just (FeatureNameModification newName)
                 )
```

The calculations for groups follow a very similar approach.

```haskell
calculateGroupModifications ::
  Map GroupId FlatGroup ->
  Map GroupId FlatGroup ->
  Map GroupId GroupModification
calculateGroupModifications =
  Merge.merge
    (Merge.mapMissing (const inPrev))
    (Merge.mapMissing (const inNew))
    (Merge.zipWithMaybeMatched (const inBoth))
  where
    inPrev _ = GroupRemove
    inNew (FlatGroup parent groupType) =
      GroupAdd parent groupType
    inBoth prev new =
      let FlatGroup prevParent prevType = prev
          FlatGroup newParent newType = new
       in if prev == new
            then Nothing
            else Just $
              GroupModification
                ( if prevParent == newParent
                    then Nothing
```

```
                    else Just (GroupParentModification
                    ↪  newParent)
                )
                ( if prevType == newType
                    then Nothing
                    else Just (GroupTypeModification newType)
                )
```

To visualize how this process is handled, we will look at two feature models. The two feature models are passed to the `diffFeatureModels` function, which derives the modifications. As described in the code above, the function compares the features and groups in both feature models, in order to derive the modifications representing the changes between the two. A visualization of the input and output of the algorithm can be seen in Figure 4.3.

Previous Feature Model   **Current Feature Model**

```
F1                                    F1
        Features                              Features
   ("f1", (-, Mandatory, "F1"))          ("f1", (-, Mandatory, "F1"))
   ("f2", ("g1", Mandatory, "F2"))       ("f3", ("g1", Optional, "F3"))

∧                                     ∨
        Groups                                Groups
      ("g1", ("f1", And))                  ("g1", ("f1", Or))

F2                                    F3
```

Feature And Group Modifications Derived

```
                    Features
            ("f2", FeatureRemove)
    ("f3", FeatureAdd ("g1", Optional, "F3")
                    Groups
  ("g1", GroupModification - (GroupTypeModification Or))
```

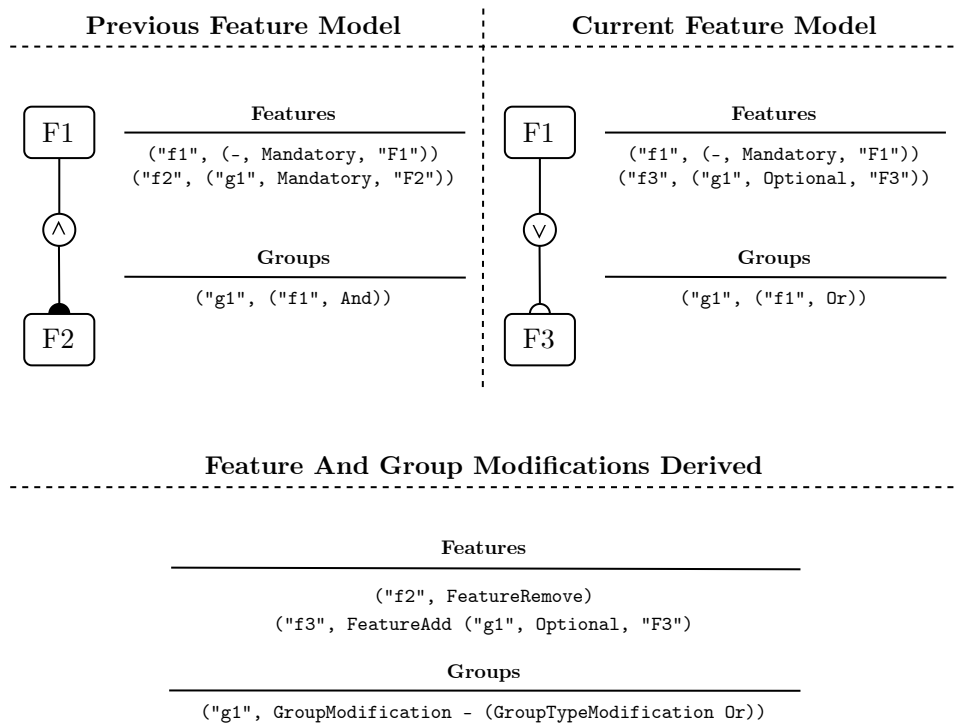Figure 4.3: Result of executing `diffFeatureModels` on two feature models

In this example, we can see that the feature with ID `f1` was unchanged, `f2` got removed and `f3` was added. As for our only group, `g1`, the type field of the group changed from `And` to `Or`. Although a bit simplified, the `merge` function combines the features, `prev` and `curr` in the following way:

```
prev   = [("f1", prevF1), ("f2", prevF2)]
```

```
curr   = [("f1", currF1),                    ("f3", currF3)]
result = [ ("f1", inBoth prevF1 currF1)
         , ("f2", inPrev prevF2)
         , ("f3", inNew currF3)
         ]
```

Since both `prevF1` and `currF1` are equal, `inBoth` will return `Nothing`, representing that we do not need a modification. As for f2 and f3, `inPrev` and `inNew` will return `FeatureRemove` and `FeatureAdd` respectively.

```
result = [ ("f1", Nothing)
         , ("f2", FeatureRemove)
         , ("f3", FeatureAdd ...)
         ]
```

In the case of feature `f1`, since the feature appears in both feature models, we use the merge tactic `zipWithMaybeMatched`, which allows us to filter out results we do not want in the final mapping. Since we are not concerned about equal features, returning `Nothing` will tell the function to filter out the result.

The final result for our feature modification are the following:

```
result = [("f2", FeatureRemove), ("f3", FeatureAdd ...)]
```

As for our groups, we only have one, namely g1. Aligning and comparing our group results in the following:

```
prev   = [("g1", prevGroup)]
curr   = [("g1", currGroup)]
result = [("g1", inBoth prevGroup currGroup)]
```

Since the two groups are unequal, the result would be a `GroupModification` on the type field. By wrapping the modification in a `Just`, we are telling the merge tactic to include this result in the final mapping.

```
result = [("g1", Just $ GroupModification Nothing
                    (GroupTypeModification Or))]
```

After applying the merge tactic `zipWithMaybeMatched`, we achieve our final result for the group modifications:

```
result = [("g1", GroupModification Nothing
                    (GroupTypeModification Or))]
```

4.3 Detecting the Changes Between Versions

Up until this point, we have only considered transformations on single evolution plans. We have created a method for transforming an evolution plan into a merge-ready evolution plan using our two functions `flatten-EvolutionPlan` and `deriveModifications`. However, since we want to merge two evolution plans into a single evolution plan, we have to figure out an approach to detect, compare and merge the changes into a single unified evolution plan.

The process of merging two different versions of an evolution plan involves figuring out what changes each version has made. In order to do so, we will utilize the common evolution plan both version were derived from, in order to confidently tell what changes has been made to each plan. As seen in our outline for the three-way merge algorithm in Figure 3.2 on page 24, we will transform all three evolution plans into the `FlatModificationEvolutionPlan` representation, then attempt to merge version 1 and 2 with respect to the base evolution plan.

4.3.1 Extending the Simple Three-Way Merge Example

Before we attempt to explain how the changes between the versions are detected and represented, we present an example consisting of three evolution plans. To create this three-way merge example, we will revisit our simple evolution plan example from in Section 4.1.2 on page 32. The simple evolution plan presented will act as our base evolution plan. Using the base evolution plan, we will create two evolution plans, version 1 and version 2, which is derived from the common evolution plan. The changes to the derived evolution plans include the following:

- **Version 1**: Includes two changes to the base evolution plan: (1) Adding a feature `F4` to the and-group at time point 1. (2) At time point 2, there was originally scheduled a group type modification from `And` to `Or`, but this version changed the modification to create a `Alternative` group instead.

- **Version 2**: Includes only one change to the base evolution plan. At time point 2, there was originally scheduled a renaming of the root feature. However, in this version, the scheduled renaming was removed.
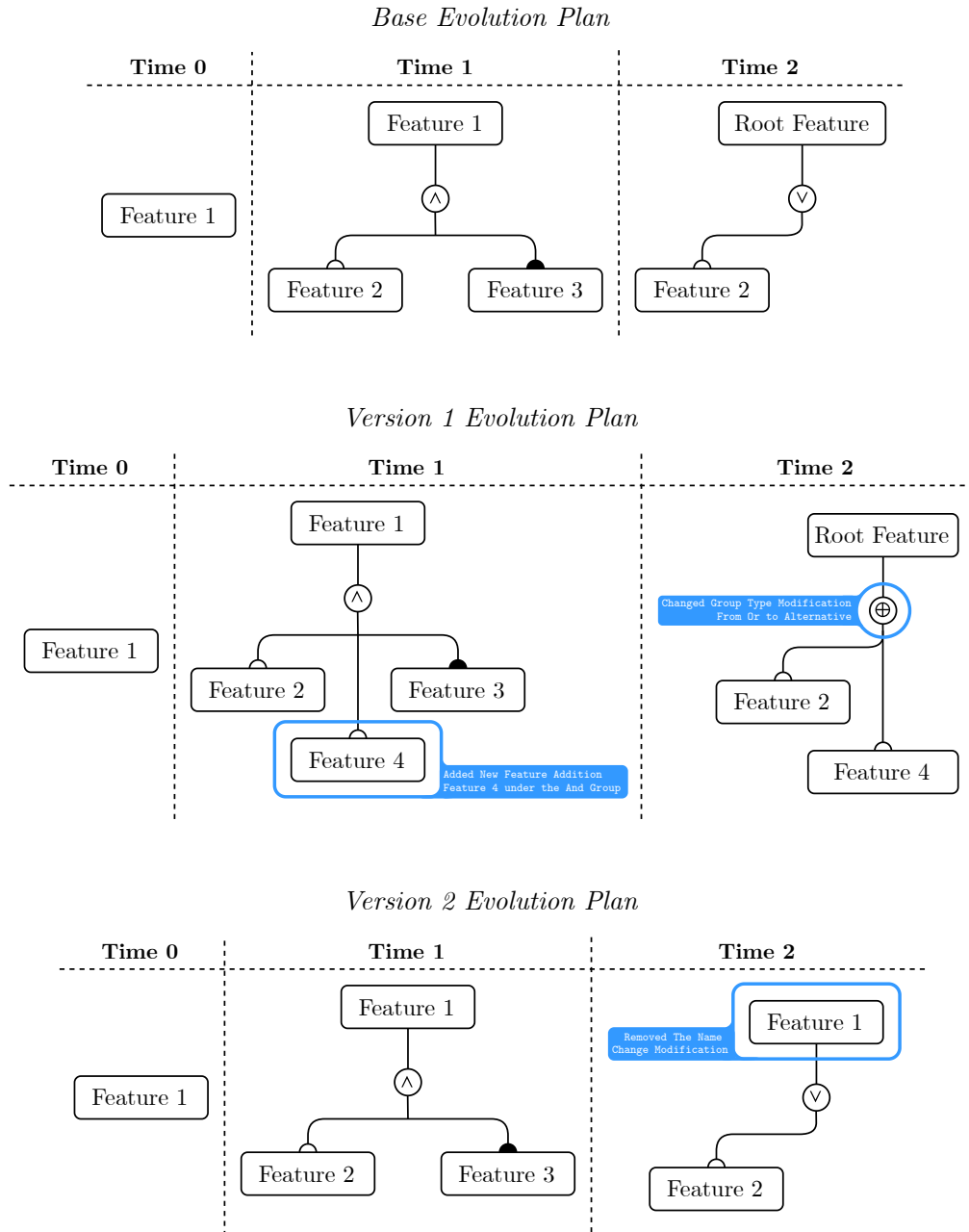
The three evolution plans is visualized in Figure 4.4.

Figure 4.4: A simple example of three evolution plans, as input of the three-way merge algorithm

4.3.2 Representing Changes Between Versions

Before attempting to merge the different versions, we want to detect what *changes* have been made in both derived versions. We present data types for representing such changes.

An important aspect to notice is the difference between *modifications* and *changes*. With modifications, we are talking about the changes between two feature models. The modifications are a part of the evolution plan, and has nothing to do with detecting changes between different versions. However, changes represent the actual changes that have been done to the base evolution plan in one of the derived versions.

With our changes, we are not adding, removing or changing features and groups, but rather adding, removing or changing the modifications themselves. The changes are working on a meta-level, allowing us to represent changes to the modifications. This distinction is a subtle, yet important factor. If one of the derived versions is removing a feature at a time point, this is actually represented as an addition. This is because the feature removal is represented as a new modification in the evolution plan, and we represent this by saying we "add" a new modification, which is the feature removal.

We present an evolution plan representing the merging of all three evolution plans. This representation follow a similar structure to our `FlatModificationEvolutionPlan`, which is why we reuse the data type `TransformationEvolutionPlan` defined in Section 4.2.3 on page 43.

```
type MergeEvolutionPlan featureModel =
  TransformationEvolutionPlan DiffResult featureModel
```

The transformations between each time points are no longer represented as `Modifications`, but rather `DiffResult`. This data type represents the union of all the modifications from all three versions, sorted and organized to our needs. Each single feature and group modification are merged based on their ID. The result of merging the three different modifications for a single feature or group is represented in `SingleDiffResult`, which can have three different outcomes:

- **NoChange**: A modification existed in the base version, and was not changed or removed in either version.

- **ChangedInOne**: A modification *changed* in one of the derived versions. This includes three scenarios; (1) a modification did not exist in the base, and were added in the derived version, (2) a modification existed in the base, but were removed in the derived version, and (3) a modification from the base were changed to another modification in the derived.

54

- **ChangedInBoth**: A modification changed in both versions. Similar to ChangedInOne, this includes changes where a modification existed in base and modifications that did not exist in the base.

The data types related to `DiffResult` are as follows:

```
data DiffResult = DiffResult
  { features :: Map FeatureId FeatureDiffResult
  , groups :: Map GroupId GroupDiffResult
  }

type FeatureDiffResult =
  SingleDiffResult FeatureModification

type GroupDiffResult =
  SingleDiffResult GroupModification

data SingleDiffResult modificationType
  = NoChange modificationType
  | ChangedInOne Version (OneChange modificationType)
  | ChangedInBoth (BothChange modificationType)

data OneChange modificationType
  = OneChangeWithBase
      modificationType -- Base
      (RemovedOrChangedModification modificationType)
        -- ^ Derived (V1 or V2)
  | OneChangeWithoutBase
      (AddedModification modificationType)
        -- ^ Derived (V1 or V2)

data BothChange modificationType
  = BothChangeWithBase
      modificationType -- Base
      (RemovedOrChangedModification modificationType) -- V1
      (RemovedOrChangedModification modificationType) -- V2
  | BothChangeWithoutBase
      (AddedModification modificationType) -- V1
      (AddedModification modificationType) -- V2

data RemovedOrChangedModification modificationType
  = RemovedModification
  | ChangedModification modificationType

data AddedModification modificationType
  = AddedModification modificationType
```

```haskell
data Version
  = V1 | V2
```

4.3.3   Representing the Extended Example

Revisiting our example from Figure 4.4, we can now represent the changes with our newly defined data types. Our merged version of the evolution plan is as follows:

```haskell
simpleExampleMergedPlan :: MergeEvolutionPlan FlatFeatureModel
simpleExampleMergedPlan =
  TransformationEvolutionPlan
    0
    initial
    [Plan 1 diffResult1, Plan 2 diffResult2]
  where
    initial =
      FlatFeatureModel
        "rootFeature"
        [
          ( "rootFeature"
          , FlatFeature Nothing Mandatory "Feature 1"
          )
        ]
        []
    diffResult1 =
      DiffResult
        [ ("feature2", NoChange
            (FeatureAdd "group" Optional "Feature 2"))
        , ("feature3", NoChange
            (FeatureAdd "group" Mandatory "Feature 3"))
        , ("feature4", ChangedInOne V1 (OneChangeWithoutBase
            (AddedModification
              (FeatureAdd "group" Optional "Feature 4"))))
        ]
        [("group", NoChange
            (GroupAdd "rootFeature" And))]
    diffResult2 =
      DiffResult
        [ ("feature3", NoChange FeatureRemove)
        , ("rootFeature", ChangedInOne V2 (OneChangeWithBase
            (FeatureModification Nothing Nothing
                (Just (FeatureNameModification "Root Feature")))
```

```
                RemovedModification))
      ]
      [ ("group", ChangedInOne V1 (OneChangeWithBase
          (GroupModification Nothing
            (Just (GroupTypeModification Or)))
          (ChangedModification (GroupModification Nothing
            (Just (GroupTypeModification Alternative))))))
      ]
```

Since most of the modifications remain unchanged, they are represented on the form ("id", NoChange modification). However, for our two changes in version 1, and one change in version 2, the changes are represented as ("id", ChangedInOne version change). Since we have no overlapping changes in the derived versions, we have no ChangedInBoth changes.

4.3.4 Calculating the Changes

In order to merge the three evolution plans into one evolution plan representing all the changes from the different versions, we define a function createMergePlan. Creating this representation requires unifying every time point in all three evolution plans. For each time point, the different modifications are combined into our new representation DiffResult.

```
createMergePlan ::
  FlatModificationEvolutionPlan ->
  FlatModificationEvolutionPlan ->
  FlatModificationEvolutionPlan ->
  MergeEvolutionPlan FlatFeatureModel
createMergePlan base v1 v2 =
  base & L.plans
    %~ \basePlans ->
      mergePlans
        basePlans
        (v1 ^. L.plans)
        (v2 ^. L.plans)

mergePlans ::
  [Plan Modifications] ->
  [Plan Modifications] ->
  [Plan Modifications] ->
  [Plan DiffResult]
mergePlans basePlans v1Plans v2Plans =
```

```haskell
  mergePlansWithTimes
    (collectAllTimePoints basePlans v1Plans v2Plans)
    basePlans
    v1Plans
    v2Plans

mergePlansWithTimes ::
  [Time] ->
  [Plan Modifications] ->
  [Plan Modifications] ->
  [Plan Modifications] ->
  [Plan DiffResult]
mergePlansWithTimes [] _ _ _ = []
mergePlansWithTimes (time : times) basePlans v1Plans v2Plans =
  Plan
    time
    ( diffModifications
        baseModifications
        v1Modifications
        v2Modifications
    ) :
  mergePlansWithTimes
    times
    nextBasePlans
    nextV1Plans
    nextV2Plans
  where
    (baseModifications, nextBasePlans) =
      getModificationForTime basePlans time
    (v1Modifications, nextV1Plans) =
      getModificationForTime v1Plans time
    (v2Modifications, nextV2Plans) =
      getModificationForTime v2Plans time
```

In some cases, one of the versions might introduce new time points. The new time points might be added at the end of the base evolution plan, or sometimes added somewhere in the middle of the existing plan. To handle this we define a function, `collectAllTimePoints`, for combining and collecting the time points for all the plans. We create `getModification-ForTime`, which returns the modifications for a given time. Since the different plans do not necessarily include all the same time points, the function will create an empty list of modifications when a time point is not present in the given evolution plan.

```haskell
collectAllTimePoints ::
  [Plan a] ->
```

```
    [Plan a] ->
    [Plan a] ->
    [Time]
collectAllTimePoints basePlans v1Plans v2Plans =
  merge (merge baseTimes v1Times) v2Times
  where
    baseTimes = basePlans ^.. traversed . L.timePoint
    v1Times = v1Plans ^.. traversed . L.timePoint
    v2Times = v2Plans ^.. traversed . L.timePoint
    merge (x : xs) (y : ys)
      | x == y = x : merge xs ys
      | x < y = x : merge xs (y : ys)
      | otherwise = y : merge (x : xs) ys
    merge xs ys = xs ++ ys

getModificationForTime ::
  [Plan Modifications] ->
  Time ->
  (Modifications, [Plan Modifications])
getModificationForTime [] _ = (emptyModifications, [])
getModificationForTime plans time =
  let Plan planTime modification : rest = plans
   in if time == planTime
         then (modification, rest)
         else (emptyModifications, plans)

emptyModifications :: Modifications
emptyModifications = Modifications M.empty M.empty
```

The `diffModifications` function defined below handles the transformation combining modifications into the `DiffResult` type. Combining the modifications of groups follows the same general process as with feature modifications.

Combining the different `Maps` follows a similar approach as deriving the modifications between feature models (See Section 4.2.3 on page 46), using the `merge` function from `Data.Map.Merge`[3]. Since we are merging three maps instead of two, we combine the three maps in two steps; (1) We combine the two derived modifications into a intermediate result using the function `mergeDerived`, and (2), we combine the base modifications with the combined derived result using `mergeBaseAndDerived`.

```
diffModifications ::
  Modifications ->
```

---

[3]`https://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map-Merge-Strict.html`

```haskell
  Modifications ->
  Modifications ->
  DiffResult
diffModifications base v1 v2 =
  DiffResult
    ( mergeMaps
        (base ^. L.features)
        (v1 ^. L.features)
        (v2 ^. L.features)
    )
    ( mergeMaps
        (base ^. L.groups)
        (v1 ^. L.groups)
        (v2 ^. L.groups)
    )
  where
    mergeMaps baseMap v1Map v2Map =
      mergeBaseAndDerived
        baseMap
        $ mergeDerived v1Map v2Map

mergeBaseAndDerived ::
  (Ord a, Eq modification) =>
  M.Map a modification ->
  M.Map a (DerivedComparisionResult modification) ->
  M.Map a (SingleDiffResult modification)
mergeBaseAndDerived =
  Merge.merge
    (Merge.mapMissing (const inBase))
    (Merge.mapMissing (const inDerived))
    (Merge.zipWithMatched (const inBoth))
  where
    inBase baseMod = withBase baseMod Nothing Nothing
    inDerived derivedResult =
      case derivedResult of
        OneVersion version mod ->
          ChangedInOne
            version
            (OneChangeWithoutBase (AddedModification mod))
        BothVersions v1Mod v2Mod ->
          ChangedInBoth
            ( BothChangeWithoutBase
                (AddedModification v1Mod)
                (AddedModification v2Mod)
            )
```

```haskell
    inBoth baseMod derivedResult =
      case derivedResult of
        OneVersion V1 mod ->
          withBase baseMod (Just mod) Nothing
        OneVersion V2 mod ->
          withBase baseMod Nothing (Just mod)
        BothVersions v1Mod v2Mod ->
          withBase baseMod (Just v1Mod) (Just v2Mod)
    withBase baseMod mV1Mod mV2Mod =
      case (Just baseMod /= mV1Mod, Just baseMod /= mV2Mod) of
        (True, True) ->
          ChangedInBoth
            ( BothChangeWithBase
                baseMod
                (removeOrChanged mV1Mod)
                (removeOrChanged mV2Mod)
            )
        (True, False) ->
          ChangedInOne
            V1
            ( OneChangeWithBase
                baseMod
                (removeOrChanged mV1Mod)
            )
        (False, True) ->
          ChangedInOne
            V2
            ( OneChangeWithBase
                baseMod
                (removeOrChanged mV2Mod)
            )
        (False, False) -> NoChange baseMod
    removeOrChanged Nothing = RemovedModification
    removeOrChanged (Just mod) = ChangedModification mod

data DerivedComparisionResult modification
  = OneVersion Version modification
  | BothVersions modification modification

mergeDerived ::
  Ord a =>
  M.Map a modification ->
  M.Map a modification ->
  M.Map a (DerivedComparisionResult modification)
mergeDerived =
```

```
Merge.merge
  (Merge.mapMissing (const (OneVersion V1)))
  (Merge.mapMissing (const (OneVersion V2)))
  (Merge.zipWithMatched (const BothVersions))
```

4.4 Merging Intended Changes

Now that we have a representation for the unification of all three evolution plans, we can begin the process of creating a single merged evolution plan. We created the type `MergeEvolutionPlan` for having a representation for all three evolution plans, and our `createMergePlan` for transforming the three evolution plans to this representation. We will now present the next step of our algorithm, `unifyMergePlan`, which joins the different modifications in the `MergeEvolutionPlan` and produces a single, unified `FlatModificationEvolutionPlan`.

In the three-way merge algorithm outline from Figure 3.2 on page 24, the `mergePlan` phase takes three evolution plans and merge them into a single evolution plan. In the actual implementation, we have split this up in two phases, `createMergePlan` and `unifyMergePlan`. This has several benefits. Both functions have clearly defined purposes. The purpose of `createMergePlan` is to create a better representation for all the modifications, making it easier to see what changes each version made to the base. Having an intermediate representation like `Merge-EvolutionPlan` also serves as documentation, letting the readers see more clearly what cases have to be considered. This benefit is also present in our `unifyMergePlan` algorithm defined below, since we now can see what modification the algorithm chooses, which it discards, and which combinations result in errors.

4.4.1 Merge Conflicts

In our three-way merge algorithm, unifying the merge plan is our first point of potential failure. As mentioned in the section discussing conflicts (Section 3.3.2 on page 25), merging the efforts into a single evolution plan might result in *merge* conflicts. A merge conflict could arise due to diverging changes for a single feature or group. This can only happen when a change is present in both derived versions. Modeling our potential conflicts are done in the following way:

```
data Conflict
  = Merge Time MergeConflict
```

```
  | Local Time LocalConflict
  | Global Time GlobalConflict
```

All of our different conflict types store the time in which the error occured, as well as information specific to the given conflict. Since merging the plan could only raise a merge conflict, we will get into the detail of local and global conflicts later. We define `MergeConflict` as follows:

```
data MergeConflict
  = FeatureConflict FeatureId (BothChange FeatureModification)
  | GroupConflict GroupId (BothChange GroupModification)
```

Notice that we reuse the `BothChange` from our definition of `Merge-EvolutionPlan`. This contains all information about what modifications were originally in the base evolution plan, as well as what changes were made in both versions.

Propagating the errors

As we will see in the definitions below, the `unifyMergePlan` algorithm takes our `MergeEvolutionPlan` as an argument, and returns an `Either Conflict FlatModificationEvolutionPlan`. Returning an `Either` means that the algorithm will either succeed, which will yield a `Right flatModEP`, or it will fail, yielding a `Left conflict`. The function attempts to unify every time point, change and modification into our `FlatModificationEvolutionPlan`. If it produces a merge conflict somewhere, the conflict automatically propagates upwards until the entire merge algorithm results in a conflict.

To achieve this without to much bloat, we will leverage Haskell's powerful type system and syntactic abstractions. Using functions such as `%%~` and `M.traverseMaybeWithKey`, as well as syntactic abstractions like do-notation, we can let the conflicts and errors automatically propagate once they occur. We will not go into great detail about how this works, but it is beneficial to know that once a conflict occurs, it is propagated to the top level.

4.4.2 Specifying the Unification of the Merge Plan

The first three functions look at the plans for each time point. Each time point contains information about the modifications of features and groups, as well as how each derived version changes these modifications. The functions go through each feature and group, and call the `unifySingle-DiffResult` to unify the changes to a single feature or group.

```haskell
unifyMergePlan ::
  MergeEvolutionPlan FlatFeatureModel ->
  Either Conflict FlatModificationEvolutionPlan
unifyMergePlan =
  L.plans . traversed %%~ unifyTimePointResult

unifyTimePointResult ::
  Plan DiffResult ->
  Either Conflict (Plan Modifications)
unifyTimePointResult (Plan time (DiffResult fs gs)) = do
  fs' <- unifyModificationsMap FeatureConflict time fs
  gs' <- unifyModificationsMap GroupConflict time gs
  return $ Plan time (Modifications fs' gs')

unifyModificationsMap ::
  Eq modificationType =>
  (modIdType -> BothChange modType -> MergeConflict) ->
  Time ->
  M.Map modIdType (SingleDiffResult modType) ->
  Either Conflict (M.Map modIdType modType)
unifyModificationsMap checkBothOverlapping timePoint =
  M.traverseMaybeWithKey
    (unifySingleDiffResult checkBothOverlapping timePoint)
```

The main work is done converting a `SingleDiffResult` to a `modification-Type`. This function is general and works for both features and groups, meaning we will get either a `FeatureModification` or a `Group-Modification`. This function will fail with a conflict if two changes from the derived versions cannot be unified. If it succeeds, the function can either return `Nothing`, indicating that an modification were removed and should not occur in the merged evolution plan, or return `Just modification` if a modification should occur in the merged plan.

```haskell
unifySingleDiffResult ::
  Eq modType =>
  (modIdType -> BothChange modType -> MergeConflict) ->
  Time ->
  modIdType ->
  SingleDiffResult modType ->
  Either Conflict (Maybe modType)
unifySingleDiffResult conflictHandler time id diffResult =
  case diffResult of
    NoChange baseMod ->
      Right (Just baseMod)
    ChangedInOne version (OneChangeWithBase baseMod
      ↪  RemovedModification) ->
```

```
      Right Nothing
    ChangedInOne version (OneChangeWithBase baseMod
    ↪ (ChangedModification derivedMod)) ->
      Right (Just derivedMod)
    ChangedInOne version (OneChangeWithoutBase
    ↪ (AddedModification derivedMod)) ->
      Right (Just derivedMod)
    ChangedInBoth bothChange ->
      checkOverlappingChanges
        conflictHandler
        time
        id
        bothChange
```

In case both versions change a feature or group modification, we only
want to raise an error if the changes are different.

```
checkOverlappingChanges ::
  Eq modType =>
  (modIdType -> BothChange modType -> MergeConflict) ->
  Time ->
  modIdType ->
  BothChange modType ->
  Either Conflict (Maybe modType)
checkOverlappingChanges conflictHandler time id bothChange =
  case bothChange of
    BothChangeWithoutBase (AddedModification v1)
    ↪ (AddedModification v2) ->
      ensureNotConflicting v1 v2
    BothChangeWithBase base RemovedModification
    ↪ RemovedModification ->
      Right Nothing
    BothChangeWithBase base (ChangedModification v1)
    ↪ (ChangedModification v2) ->
      ensureNotConflicting v1 v2
    BothChangeWithBase{} ->
      conflict
  where
    conflict = Left (Merge time (conflictHandler id
    ↪ bothChange))
    ensureNotConflicting v1Modification v2Modification =
      if v1Modification == v2Modification
        then Right (Just v1Modification)
        else conflict
```

4.4.3 Resulting Evolution Plan After Merging the Example

Looking at our running example, visualized in Figure 4.4 on page 53, the result of merging the three plans is successful. Since none of the changes from each version overlap, no merge conflict was produced either. Each change in both versions was included, and the result was the following:

```
simpleExampleUnifiedPlan
  :: Either Conflict FlatModificationEvolutionPlan
simpleExampleUnifiedPlan =
  Right $
    TransformationEvolutionPlan
      0
      initial
      [ Plan 1 modifications1
      , Plan 2 modifications2
      ]
  where
    initial =
      FlatFeatureModel
        "rootFeature"
        [
          ( "rootFeature"
          , FlatFeature Nothing Mandatory "Feature 1"
          )
        ]
        []
    modifications1 =
      Modifications
        [
          ( "feature2"
          , FeatureAdd "group" Optional "Feature 2"
          )
        ,
          ( "feature3"
          , FeatureAdd "group" Mandatory "Feature 3"
          )
        ,
          ( "feature4"
          , FeatureAdd "group" Optional "Feature 4"
          )
        ]
        [("group", GroupAdd "rootFeature" And)]
    modifications2 =
      Modifications
```

```
    [("feature3", FeatureRemove)]
    [
      ( "group"
      , GroupModification
          Nothing
          (Just (GroupTypeModification Alternative))
      )
    ]
```

The result of the `unifyMergePlan` function returns what we want. From both derived versions, there were a total of three changes. A new feature was added at time 1, the name change in time 2 was removed and the group type modification in time 2 was changed to an `Alternative` group instead of `Or` group.

4.5 Ensuring Structural and Semantic Soundness

Now that we have defined a method of detecting and merging changes to evolution plans, we have to check that the resulting evolution plan results in a valid and sound evolution plan, keeping both the structure and semantics in tact.

To ensure soundness, we design an algorithm that makes sure that each modification is valid. In order to do so, we convert our `Flat-ModificationEvolutionPlan` representation back to our normal form for evolution plans, namely `TreeUserEvolutionPlan`. Doing so lets us see the effects of applying each modification, and ensures that the resulting feature models and evolution plan is correct.

Looking back at the algorithm outline in Figure 3.2 on page 24, we can see that there are three remaining steps in the algorithm; `integrate-Modifications`, `checkModifications` and `unflattenEvolutionPlan`. First, we will take the unchecked evolution plan and integrate every modification for each time point. Next, we will check that the resulting feature models are following the structural and semantic constraints of evolution plans. Lastly, we will convert the sound evolution plan back to our tree-based normal form for evolution plans. Each of these steps are discussed in more detail in Section 4.5.1, Section 4.5.5 and Section 4.5.6.

Integrating and Checking Modifications In reality, the two steps `integrateModifications` and `checkModifications` are more tightly integrated than visualized in the outline. Instead, we have a single algo-

rithm for doing both things, `integrateAndCheckModifications`. By starting with the initial feature model and the first time point, the function will first apply every modification to the feature model, then check that the result is sound. With the resulting feature model, we will take the next time point and do the process all over again. This will continue until we have a list of feature models that are checked for soundness.

For a single time point, we need to apply all the modifications before checking for soundness. This is due to us having no specific ordering of operations, and the feature model might be temporarily invalid while applying modifications. For this reason, we do not check constraints immediately after applying a modification. Instead, we note the potential points where our result might be invalid, and check for correctness after every modification for a given time point has been integrated.

When applying modifications, we generate a list of *dependencies* that we later pass onto `checkModifications`. What dependencies arise will depend on the modification at hand, but typically these dependencies include things like checking for cycles, non-existing parent relations or well-formedness constraints. These dependencies along with the current feature model are then passed onto `checkModifications`, which checks every dependency and either accepts or rejects the feature model.

Below, we define the code necessary to intertwine both the integration and soundness checking of modifications. Since integrating and checking modifications can both lead to conflicts, the functions return `Either Conflict value` instead of just `value`. Using do-notation, the monadic structure makes sure the error is propagated immediately if a conflict occurs.

```
integrateAndCheckModifications ::
  FlatModificationEvolutionPlan ->
  Either Conflict FlatUserEvolutionPlan
integrateAndCheckModifications evolutionPlan =
  case evolutionPlan of
    TransformationEvolutionPlan initialTime initialFM plans ->
      UserEvolutionPlan
        <$> scanEvolutionPlan
          plans
          (TimePoint initialTime initialFM)


scanEvolutionPlan ::
  [Plan Modifications] ->
  TimePoint FlatFeatureModel ->
  Either Conflict [TimePoint FlatFeatureModel]
scanEvolutionPlan [] timePoint = return [timePoint]
scanEvolutionPlan (plan : plans) currentTimePoint = do
```

```
(nextTimePointUnchecked, dependencies) <-
  runWriterT $ integrateSinglePlan plan currentTimePoint
nextTimePoint <-
  checkGlobalConflict dependencies nextTimePointUnchecked
convertedEvolutionPlan <-
  scanEvolutionPlan plans nextTimePoint
return $ currentTimePoint : convertedEvolutionPlan
```

As discussed, the first part integrates the modifications. Most of the work is here done by `integrateSinglePlan`, which returns a `WriterT [Dependency] (Either Conflict) (TimePoint FlatFeatureModel)`. Using the `Writer` and `Either` monad, we can write a function handling conflicts, writing dependencies and returning the merged time point without too much boilerplate. This is made possible by the `WriterT` monad transformer, which allows us to compose monads. In this case, we have our `Either Conflict` monad, which lets us propagate errors. As we integrate modifications one by one, we also generate a list of dependencies, which the `Writer` monad lets us do without much boilerplate. These two monads are then combined, allowing us to use the `runWriterT` functions which return both the next time point and the dependencies that need to be checked in an `Either` environment that propagates errors when they occur.

The generated dependencies and the next time point are then passed to `checkGlobalConflict`, which either succeeds or raises a conflict. Upon failure, the do-notation and monadic structure propagate the error. However, if it succeeds, the rest of the time points are recursively called.

4.5.1   Applying Every Modification in the Evolution Plan

In order to integrate every single modification for a single time point, the `integrateSinglePlan` function is called. As discussed, the modifications for a single time point have no ordering, so we can arbitrarily choose an application ordering. The modifications are applied by calling either `integrateFeature` or `integrateGroup` with the feature model, which will return the feature model with the modification applied. The resulting feature model is then passed to the next modification. The process is continued until all modifications are applied.

The core idea of this is the `foldl` function, which has the following signature for lists: `(fm -> mod -> fm) -> fm -> [mod] -> fm`. Using it for our purpose, it will apply the modifications on the feature model as discussed. However, we will use the more complicated variant, `ifoldlMOf` instead. The main reason is that it allows us to fold in a monadic context. In our instance, our monad is both the `Writer` and `Either` monad. This

means that if an error occurs somewhere, the computation stops and the conflict is returned. The `Writer` monad lets us append dependencies without actually worrying about passing the list of dependencies as argument and returning it from the functions.

```
integrateSinglePlan ::
  Plan Modifications ->
  TimePoint FlatFeatureModel ->
  WriterT
    [Dependency]
    (Either Conflict)
    (TimePoint FlatFeatureModel)
integrateSinglePlan
  (Plan nextTime modifications)
  (TimePoint _ featureModel) =
    TimePoint nextTime <$> newFeatureModel
    where
      newFeatureModel =
        integrateFeatures featureModel >>= integrateGroups
      integrateFeatures fm =
        ifoldlMOf
          (L.features . itraversed)
          (integrateFeature nextTime)
          fm
          modifications
      integrateGroups fm =
        ifoldlMOf
          (L.groups . itraversed)
          (integrateGroup nextTime)
          fm
          modifications
```

The `integrateFeature` and `integrateGroup` functions is given a modification on a feature or group, as well as the current feature model. Based on the type of modification, the function writes dependencies with the `tell` function and incorporates the modification in the feature model. If a conflict occurs somewhere, the `throwError` function is used to short circuit the function and return the conflict to the top level.

```
integrateFeature ::
  Time ->
  FeatureId ->
  FlatFeatureModel ->
  FeatureModification ->
  WriterT [Dependency] (Either Conflict) FlatFeatureModel
integrateFeature time featureId fm featureMod =
```

```
case featureMod of
  FeatureAdd parentGroupId featureType name ->
    case M.lookup featureId (fm ^. L.features) of
      Nothing -> do
        tell
          . fmap (FeatureDependency featureMod)
          $ [ ParentGroupExists parentGroupId
            , UniqueName name
            , FeatureIsWellFormed featureId
            ]
        return $
          fm
            & L.features
              . at featureId
              ?~ FlatFeature
                (Just parentGroupId)
                featureType
                name
      Just oldFeature ->
        throwError $
          Local
            time
            (FeatureAlreadyExists featureMod featureId)
  FeatureRemove ->
    case M.lookup featureId (fm ^. L.features) of
      Nothing ->
        throwError $
          Local
            time
            (FeatureNotExists featureMod featureId)
      Just oldFeature -> do
        tell . fmap (FeatureDependency featureMod) $
          [NoChildGroups featureId]
        return $ fm & L.features . at featureId .~ Nothing
  FeatureModification parentIdMod featureTypeMod nameMod ->
    if has (L.features . ix featureId) fm
      then
        pure fm
          >>= integrateParentMod
          >>= integrateTypeMod
          >>= integrateNameMod
      else
        throwError $
          Local time (FeatureNotExists featureMod featureId)
    where
```

```haskell
integrateParentMod ::
  FlatFeatureModel ->
  WriterT
    [Dependency]
    (Either Conflict)
    FlatFeatureModel
integrateParentMod fm =
  case parentIdMod of
    Nothing -> return fm
    Just (FeatureParentModification newValue) -> do
      tell . fmap (FeatureDependency featureMod) $
        [ ParentGroupExists newValue
        , NoCycleFromFeature featureId
        , FeatureIsWellFormed featureId
        ]
      return $
        fm
          & L.features
            . ix featureId
            . L.parentGroupId
            ?~ newValue

integrateTypeMod ::
  FlatFeatureModel ->
  WriterT
    [Dependency]
    (Either Conflict)
    FlatFeatureModel
integrateTypeMod fm =
  case featureTypeMod of
    Nothing -> return fm
    Just (FeatureTypeModification newValue) -> do
      tell . fmap (FeatureDependency featureMod) $
        [FeatureIsWellFormed featureId]
      return $
        fm
          & L.features
            . ix featureId
            . L.featureType
          .~ newValue

integrateNameMod ::
  FlatFeatureModel ->
  WriterT
    [Dependency]
```

```haskell
                    (Either Conflict)
                    FlatFeatureModel
              integrateNameMod fm =
                case nameMod of
                  Nothing -> return fm
                  Just (FeatureNameModification newValue) -> do
                    tell . fmap (FeatureDependency featureMod) $
                      [UniqueName newValue]
                    return $
                      fm
                        & L.features
                          . ix featureId
                          . L.name
                        .~ newValue

integrateGroup ::
  Time ->
  GroupId ->
  FlatFeatureModel ->
  GroupModification ->
  WriterT [Dependency] (Either Conflict) FlatFeatureModel
integrateGroup time groupId fm groupMod =
  case groupMod of
    GroupAdd parentFeatureId groupType ->
      case M.lookup groupId (fm ^. L.groups) of
        Nothing -> do
          tell . fmap (GroupDependency groupMod) $
            [ ParentFeatureExists parentFeatureId
            ]
          return $
            fm
              & L.groups
                . at groupId
                ?~ FlatGroup parentFeatureId groupType
        Just oldGroup ->
          throwError $
          Local
            time
            (GroupAlreadyExists groupMod groupId)
    GroupRemove ->
      case M.lookup groupId (fm ^. L.groups) of
        Nothing ->
          throwError $
          Local
            time
```

73

```haskell
                (GroupNotExists groupMod groupId)
        Just oldGroup -> do
          tell . fmap (GroupDependency groupMod) $
            [NoChildFeatures groupId]
          return $ fm & L.groups . at groupId .~ Nothing
GroupModification parentFeatureIdMod groupTypeMod ->
  if has (L.groups . ix groupId) fm
    then
      pure fm
        >>= integrateParentMod
        >>= integrateTypeMod
    else
      throwError $
        Local time (GroupNotExists groupMod groupId)
  where
    integrateParentMod ::
      FlatFeatureModel ->
      WriterT
        [Dependency]
        (Either Conflict)
        FlatFeatureModel
    integrateParentMod fm =
      case parentFeatureIdMod of
        Nothing -> return fm
        Just (GroupParentModification newValue) -> do
          tell . fmap (GroupDependency groupMod) $
            [ ParentFeatureExists newValue
            , NoCycleFromGroup groupId
            ]
          return $
            fm
              & L.groups
                . ix groupId
                . L.parentFeatureId
              .~ newValue

    integrateTypeMod ::
      FlatFeatureModel ->
      WriterT
        [Dependency]
        (Either Conflict)
        FlatFeatureModel
    integrateTypeMod fm =
      case groupTypeMod of
        Nothing -> return fm
```

```
Just (GroupTypeModification newValue) -> do
  tell . fmap (GroupDependency groupMod) $
    [GroupIsWellFormed groupId]
  return $
    fm
      & L.groups
        . ix groupId
        . L.groupType
      .~ newValue
```

4.5.2   Local Conflicts

As seen in the code above, the integration of modifications to the feature
model can potentially lead to *local* conflicts. When we try to integrate the
set of modifications at a certain time point, we want to make sure that
the feature model after applying every modification is sound. This means
that we allow for the feature model to be invalid while we are under the
process of applying the modifications. However, some changes can be
guaranteed to result in an unsound feature model.

Local conflicts occur when we try to alter or remove features or groups
that do not exist, or when we try to add features that already exist. Since
our modifications are modeled as maps with IDs as keys, we guarantee
that a feature or group only has a single modification. This implies that
we cannot add and then remove a feature at the same time point. The
local conflicts can thus be reported immediately without checking the rest
of the modifications at the time point.

```
data LocalConflict
  = FeatureAlreadyExists FeatureModification FeatureId
  | FeatureNotExists FeatureModification FeatureId
  | GroupAlreadyExists GroupModification GroupId
  | GroupNotExists GroupModification GroupId
```

4.5.3   Dependencies

With local conflicts, the conflicts are local to the single feature or group at
hand. These conflicts can be raised without knowing what the rest of the
modifications are. However, some of the modifications rely on knowing
the state of other features or groups, which prevents us from reporting
these conflicts immediately. Some modifications, i.e. adding a feature,
rely on parent nodes to exist. However, to know if the parent exists or not,
we have to apply the rest of the modifications in case the parent also was

```

removed or added. For this reason, we postpone the checking until after every modification has been included, generating dependencies which mark what we need to check. The different dependencies are defined below.

```
data Dependency
  = FeatureDependency FeatureModification FeatureDependencyType
  | GroupDependency GroupModification GroupDependencyType

data FeatureDependencyType
  = NoChildGroups FeatureId
  | ParentGroupExists GroupId
  | NoCycleFromFeature FeatureId
  | FeatureIsWellFormed FeatureId
  | UniqueName String

data GroupDependencyType
  = NoChildFeatures GroupId
  | ParentFeatureExists FeatureId
  | NoCycleFromGroup GroupId
  | GroupIsWellFormed GroupId
```

Each modification yields a set of dependencies depending on what conflicts could potentially rise. This is encoded in the `integrateFeature` and `integrateGroup` functions, as well as visualized in Table 4.4.

| Modification Type | Generated Dependencies |
| --- | --- |
| Add Feature | ParentGroupExists |
| | UniqueName |
| | FeatureIsWellFormed |
| Remove Feature | NoChildGroups |
| Modify Feature Parent | ParentGroupExists |
| | NoCycleFromFeature |
| | FeatureIsWellFormed |
| Modify Feature Type | FeatureIsWellFormed |
| Modify Feature Name | UniqueName |
| Add Group | ParentFeatureExists |
| Remove Group | NoChildFeatures |
| Modify Group Parent | ParentFeatureExists |
| | NoCycleFromGroup |
| Modify Group Type | GroupIsWellFormed |

Table 4.4: Generated dependencies

### 4.5.4 Global Conflicts

The last kind of conflicts we can encounter is the *global* conflict. For a given time point, we generate a list of dependencies we need to check. If one or more dependencies are not met, we raise a global conflict. This is done by collecting the failed dependencies and returning them as a list.

```
data GlobalConflict
  = FailedDependencies [Dependency]
```

4.5.5 Checking Dependencies and Ensuring Soundness

After applying every modification for a certain time point, the generated dependencies and resulting feature model are passed to the `checkGlobal-Conflict`. The function will either succeed with the correct feature model, or fail with a list of dependencies that do not pass.

```
checkGlobalConflict ::
  [Dependency] ->
  TimePoint FlatFeatureModel ->
  Either Conflict (TimePoint FlatFeatureModel)
checkGlobalConflict dependencies tp =
  errorIfFailed (filter (not . checkDependency) dependencies)
  where
    TimePoint time featureModel = tp
    errorIfFailed failedDeps =
      case failedDeps of
        [] -> Right tp
        _ -> Left $ Global time (FailedDependencies failedDeps)
    checkDependency (FeatureDependency featureMod dType) =
      case dType of
        NoChildGroups featureId ->
          hasn't
            ( L.groups
                . traversed
                . L.parentFeatureId
                . filtered (== featureId)
            )
            featureModel
        ParentGroupExists groupId ->
          has
            (L.groups . ix groupId)
            featureModel
        NoCycleFromFeature featureId ->
```

```
                  not $ featureInCycle S.empty featureId featureModel
            FeatureIsWellFormed featureId ->
              -- If mandatory feature, parent has to be AND group
              -- === feature not mandatory or parent is and
            let featureType =
                    featureModel
                      ^?! L.features
                        . ix featureId
                        . L.featureType
                parentGroupType =
                    featureModel
                      ^?! L.parentGroupOfFeature featureId
                        . L.groupType
             in featureType /= Mandatory
                  || parentGroupType == And
          UniqueName name ->
            lengthOf
              ( L.features
                  . traversed
                  . L.name
                  . filtered (== name)
              )
              featureModel
              <= 1
checkDependency (GroupDependency groupMod dType) =
  case dType of
    NoChildFeatures groupId ->
      hasn't
        ( L.features
            . traversed
            . L.parentGroupId
            . filtered (== Just groupId)
        )
        featureModel
    ParentFeatureExists featureId ->
      has
        (L.features . ix featureId)
        featureModel
    NoCycleFromGroup groupId ->
      not $ groupInCycle S.empty groupId featureModel
    GroupIsWellFormed groupId ->
      -- Either the group is a AND group
      -- or all child features are optional
      let groupType =
              featureModel
```

```
                        ^?! L.groups
                           . ix groupId
                           . L.groupType
                    childFeatureTypes =
                       featureModel
                          ^.. L.childFeaturesOfGroup groupId
                             . L.featureType
                 in groupType == And
                       || all (== Optional) childFeatureTypes


featureInCycle ::
  S.Set (Either FeatureId GroupId) ->
  FeatureId ->
  FlatFeatureModel ->
  Bool
featureInCycle visited featureId featureModel
  | Left featureId `elem` visited = True
  | otherwise =
    case featureModel
      ^? L.features
         . ix featureId
         . L.parentGroupId
         . _Just of
      Nothing -> False -- no parent group/non existing feature
      Just parentGroupId ->
        groupInCycle
          (S.insert (Left featureId) visited)
          parentGroupId
          featureModel


groupInCycle ::
  S.Set (Either FeatureId GroupId) ->
  GroupId ->
  FlatFeatureModel ->
  Bool
groupInCycle visited groupId featureModel
  | Right groupId `elem` visited = True
  | otherwise =
    case featureModel
      ^? L.groups
         . ix groupId
         . L.parentFeatureId of
      Nothing -> False -- non existing group
      Just parentFeatureId ->
        featureInCycle
```

```
               (S.insert (Right groupId) visited)
               parentFeatureId
               featureModel
```

4.5.6 Converting Back to Normal Form

The last steps of the three-way algorithm transformed the `Flat-ModificationEvolutionPlan` back to `FlatUserEvolutionPlan`, which models the evolution plan as a list of feature models. The final and last step of the algorithm is converting this back to the `TreeUserEvolutionPlan`, which models each feature model in a recursive tree structure instead of the flat mapping based way.

This is done by retrieving the root feature and its child groups. The child groups are recursively transformed to our tree representation, and combined with the root feature to create our recursive tree structure.

```
unflattenSoundEvolutionPlan ::
  FlatUserEvolutionPlan ->
  TreeUserEvolutionPlan
unflattenSoundEvolutionPlan =
  L.timePoints
    . traversed
    %~ unflattenTimePoint


unflattenTimePoint ::
  TimePoint FlatFeatureModel ->
  TimePoint TreeFeatureModel
unflattenTimePoint (TimePoint time featureModel) =
  TimePoint time $
    TreeFeatureModel $
      unflattenFeature featureModel (featureModel ^. L.rootId)


unflattenFeature ::
  FlatFeatureModel ->
  FeatureId ->
  TreeFeature
unflattenFeature featureModel featureId =
  TreeFeature featureId featureType name childGroups
  where
    childGroupIds =
      featureModel
        ^.. L.ichildGroupsOfFeature featureId . asIndex
    childGroups =
```

```
      S.fromList $
        fmap (unflattenGroup featureModel) childGroupIds
    (FlatFeature _ featureType name) =
      featureModel ^?! L.features . ix featureId

unflattenGroup ::
  FlatFeatureModel ->
  GroupId ->
  TreeGroup
unflattenGroup featureModel groupId =
  TreeGroup groupId groupType childFeatures
  where
    childFeatureIds =
      featureModel
        ^.. L.ichildFeaturesOfGroup groupId
          . asIndex
    childFeatures =
      S.fromList $
        fmap (unflattenFeature featureModel) childFeatureIds
    (FlatGroup _ groupType) =
      featureModel ^?! L.groups . ix groupId
```

4.5.7 The Simple Example After Applying and Checking Modifications

Revisiting our simple example from Figure 4.4 on page 53, we pick up the example from where we left off. With the FlatModification-EvolutionPlan representation of the merged plan, we use integrate-AndCheckModifications to apply the modifications and check the result for soundness. This results in a list of feature models, in our FlatUser-EvolutionPlan representation. The result can be seen below.

```
simpleExampleCheckedPlan ::
  Either Conflict FlatUserEvolutionPlan
simpleExampleCheckedPlan =
  Right $
    UserEvolutionPlan
      [ TimePoint 0 fm0
      , TimePoint 1 fm1
      , TimePoint 2 fm2
      ]
  where
    fm0 =
      FlatFeatureModel
```

```
      "rootFeature"
      [
        ( "rootFeature"
        , FlatFeature
            Nothing
            Mandatory
            "Feature 1"
        )
      ]
      []
fm1 =
    FlatFeatureModel
      "rootFeature"
      [
        ( "feature2"
        , FlatFeature
            (Just "group")
            Optional
            "Feature 2"
        )
      ,
        ( "feature3"
        , FlatFeature
            (Just "group")
            Mandatory
            "Feature 3"
        )
      ,
        ( "feature4"
        , FlatFeature
            (Just "group")
            Optional
            "Feature 4"
        )
      ,
        ( "rootFeature"
        , FlatFeature
            Nothing
            Mandatory
            "Feature 1"
        )
      ]
      [
        ( "group"
        , FlatGroup
```

```
              "rootFeature"
              And
          )
      ]
    fm2 =
      FlatFeatureModel
        "rootFeature"
        [
          ( "feature2"
          , FlatFeature
              (Just "group")
              Optional
              "Feature 2"
          )
        ,
          ( "feature4"
          , FlatFeature
              (Just "group")
              Optional
              "Feature 4"
          )
        ,
          ( "rootFeature"
          , FlatFeature
              Nothing
              Mandatory
              "Feature 1"
          )
        ]
        [
          ( "group"
          , FlatGroup
              "rootFeature"
              Alternative
          )
        ]
```

4.5.8 Dependencies Generated by the Example

As we traverse the time points and apply the modifications to the current feature model, we generate dependencies that are later checked. The dependencies generated at each time point are visualized below.

```haskell
generatedDependencies :: [(Time, [Dependency])]
generatedDependencies =
  [
    ( 1
    ,
      [ FeatureDependency
          (FeatureAdd "group" Optional "Feature 2")
          (ParentGroupExists "group")
      , FeatureDependency
          (FeatureAdd "group" Optional "Feature 2")
          (UniqueName "Feature 2")
      , FeatureDependency
          (FeatureAdd "group" Optional "Feature 2")
          (FeatureIsWellFormed "feature2")
      , FeatureDependency
          (FeatureAdd "group" Mandatory "Feature 3")
          (ParentGroupExists "group")
      , FeatureDependency
          (FeatureAdd "group" Mandatory "Feature 3")
          (UniqueName "Feature 3")
      , FeatureDependency
          (FeatureAdd "group" Mandatory "Feature 3")
          (FeatureIsWellFormed "feature3")
      , FeatureDependency
          (FeatureAdd "group" Optional "Feature 4")
          (ParentGroupExists "group")
      , FeatureDependency
          (FeatureAdd "group" Optional "Feature 4")
          (UniqueName "Feature 4")
      , FeatureDependency
          (FeatureAdd "group" Optional "Feature 4")
          (FeatureIsWellFormed "feature4")
      , GroupDependency
          (GroupAdd "rootFeature" And)
          (ParentFeatureExists "rootFeature")
      ]
    )
  ,
    ( 2
    ,
      [ FeatureDependency
          FeatureRemove
          (NoChildGroups "feature3")
      , GroupDependency
```

```
        (GroupModification Nothing (Just
        ↪  (GroupTypeModification Alternative)))
        (GroupIsWellFormed "group")
    ]
  )
]
```

4.5.9 Final Output of the Simple Three-Way Example

As our example is passed through the final step, converting it back to the original representation, we get the following result.

```
simpleExampleFinalResult ::
  Either Conflict TreeUserEvolutionPlan
simpleExampleFinalResult =
  Right $
    UserEvolutionPlan
      [ TimePoint 0 fm0
      , TimePoint 1 fm1
      , TimePoint 2 fm2
      ]
  where
    fm0 =
      TreeFeatureModel $
        TreeFeature
          "rootFeature"
          Mandatory
          "Feature 1"
          []
    fm1 =
      TreeFeatureModel $
        TreeFeature
          "rootFeature"
          Mandatory
          "Feature 1"
          [ TreeGroup
              "group"
              And
              [ TreeFeature
                  "feature2"
                  Optional
                  "Feature 2"
                  []
              , TreeFeature
```

```
                    "feature3"
                    Mandatory
                    "Feature 3"
                    []
                  , TreeFeature
                    "feature4"
                    Optional
                    "Feature 4"
                    []
                ]
            ]
    fm2 =
      TreeFeatureModel $
        TreeFeature
          "rootFeature"
          Mandatory
          "Feature 1"
          [ TreeGroup
              "group"
              Alternative
              [ TreeFeature
                  "feature2"
                  Optional
                  "Feature 2"
                  []
                , TreeFeature
                  "feature4"
                  Optional
                  "Feature 4"
                  []
              ]
          ]
```

The resulting merged evolution plan is also visualized in Figure 4.5.

4.6   Efficiency of the Algorithm

In this section, we give a rough time complexity analysis of the three-way merge algorithm. How the algorithm performs is highly dependent on a number of factors, including the total amount of features and groups, the amount of modifications between each time point, as well as the similarities between the versions. To keep it simple, we perform time

Merged Evolution Plan
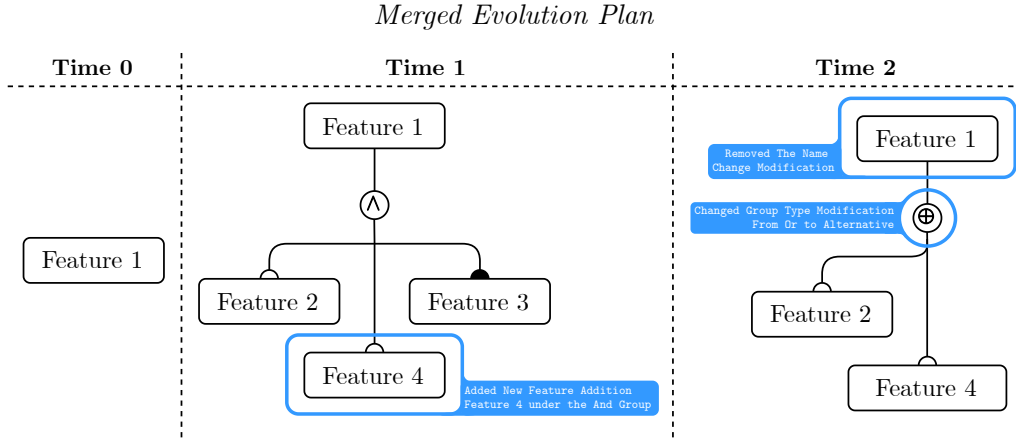
Figure 4.5: A visualization of the merged result of our simple example

complexity analysis with regards to the amount of time points, t, and the average amount of features and groups per time point, n. The analysis is performed by discussing and analyzing the complexity of each phase of the algorithm, then concluding with the total complexity of the algorithm. We will do both a worst-case analysis and an average-case analysis. The worst-case will consider how the algorithm will perform under the most unfortunate circumstances, while the average-case analysis will avoid extreme edge cases and instead consider more realistic scenarios.

Converting Individual Plans to a Flat Structure In the first function, `flattenEvolutionPlan`, we traverse the list of feature models and convert each feature model to the flat representation. Flattening each feature model requires traversing the tree structure and recursively constructing lists of `FlatFeature` and `FlatGroup`. Since list concatenation has the complexity of $O(n)$, flattening a feature model has the worst-case complexity of $O(n^2)$. Since each feature model in the evolution plan are transformed to the `FlatFeatureModel` representation, we get the worst-case complexity of $O(t \cdot n^2)$. However, we get this complexity only when we have very unbalanced trees. In very lucky cases with completely balanced trees, flattening has a complexity of $O(t \cdot n)$. In the average case, our feature models are not always perfectly balanced, which could give us a time complexity of $O(t \cdot n \cdot log(n))$.

Deriving Modifications of Individual Plans After converting each evolution plan to the flat structure, we transform each individual evolution plan to the `FlatModificationEvolutionPlan` representation using the `deriveSoundModifications` function. The conversion process is handled

by calculating the difference between every subsequent pair of feature models. Since there are t feature models, we have to calculate the difference between $t - 1$ pairs. In calculating the difference, we rely heavily on the merge function from `Data.Map.Merge`. Even though its time complexity is not noted in the documentation[4], similar functions on `Maps` such as `unionWithKey`[5] have the complexity of $O(x \cdot log(\frac{y}{x} + 1)), x \leq y$, which in our example equates to $O(n \cdot log(2)) = O(n)$. Since this is calculated $t - 1$ times per evolution plan, we get the total of $O(t \cdot n)$

Detecting Changes   With the first two functions only acting on single evolution plans, the next function we analyze takes all three evolution plans as input. Since the different versions could introduce new time points, we are using `collectAllTimePoints` to collect the different times used in all versions. This is done by simultaneously traversing all three evolution plans, requiring a complexity of $O(t_{base} + t_{v1} + t_{v2})$. For simplicity, we assume that t is the total amount of time points, which makes the complexity for `collectAllTimePoints` $O(t)$. Next, we traverse the list of time points and merge the different modifications of all three versions. To calculate the complexity, we are no longer dealing with features and feature models, but rather modifications that transform subsequent feature models. However, since we know that a single feature or group can only be modified once per time point, we know that the amount of modifications are linear with regards to the amount of nodes in a feature model. Merging the modifications of all versions for a given time point requires a time complexity of $O(n_{base} + n_{v1} + n_{v2})$, which we simplify as $O(n)$. Merging all three evolution plan thus yields a complexity of $O(t \cdot n)$.

Unifying Changes   With the `createMergePlan` function merging all three evolution plans into a single data type, `MergeEvolutionPlan`, the next step is actually merging this representation into a single evolution plan. This is done by traversing every `DiffResult` for every time point, and either raising a conflict or producing a modification. This yields a time complexity of $O(t \cdot n)$.

Integrating Modifications   Now that we have a merged evolution plan, we begin the process of ensuring soundness by applying every modification to get a list of feature models. This process is performed

---

[4]`https://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map-Merge-Strict.html`

[5]`https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Map-Strict.html`

one time point at a time, where each set of modifications is applied to the current feature model and then checked for soundness violations. Each modification is integrated one by one by folding over the set of modifications. Integrating each feature or group is mostly just lookup, remove or add operations on `Maps`, which has a complexity of $O(log(n))$. However, we are also building a list of dependencies with the `Writer` monad. Each modification only produces a maximum of five dependencies. Since the lists are concatenated using append operations, we get the unfortunate complexity of $O(n^2)$ per time point. This is a potential for improvement, where one might opt for *difference lists*[6] to achieve $O(1)$ complexity for append operations, which would give a time complexity of $O(t \cdot n)$. However, since we are just using normal lists, we get a complexity of $O(t \cdot n^2)$.

Checking for Conflicts After successfully constructing the next feature model from the previous one, the function `checkGlobalConflict` is called with the generated list of dependencies. Since there is a maximum of five dependencies generated per modification, we only have a maximum of $5n$ dependencies per time point. The complexity for checking each dependency will vary based on its type. Some dependencies, like `ParentGroupExists`, `FeatureIsWellFormed` and `ParentFeatureExists` simply simply require a lookup of a single feature or group, which has the complexity of $O(log(n))$. `NoChildGroups`, `UniqueName`, `NoChildFeatures` and `GroupIsWellFormed` requires traversing the entire set of features or groups, which yields the complexity of $O(n)$. Lastly, we have to consider `NoCycleFromFeature` and `NoCycleFromGroup`. Checking cycles requires traversing every ancestor from a given feature or group. In a fairly balanced tree, this requires $O(log(n))$ traversals, but in cases with very unbalanced trees, this might cause a complexity of $O(n)$. For each node traversed, there are two `Map` or `Set` lookups and one insertion in the `Set` of visited nodes. These are $O(log(n))$ operations, which in the worst case will result in $O(n \cdot log(n))$ per dependency. In total, the worst-case complexity of this phase is $O(t \cdot n^2 \cdot log(n))$. However, this is extremely unlikely, since it requires a very unbalanced tree and moving almost every node in every time point. In average cases, only a handful of moves are included, which would make the complexity a bit more forgiving. If we assume that there only is $log(n)$ move operations per time point, we only need to make $log(n)$ cycle checks per time point. For the normal operations, we will still have $O(n^2)$ per time point, while the move operations create a complexity of $O(n \cdot (log(n))^2) = O(n^2)$ for each time point. This assumption yields an average time complexity of $O(t \cdot n^2)$ for checking dependencies for every time point.

---

[6]`https://kseo.github.io/posts/2014-01-14-difference-listfor-writer-monad.html`

Converting Back to Original Representation The last step of the algorithm is converting from the flat feature model representation to the tree-based representation. This is done by traversing every time point and converting each feature model. Since the flat feature models stores parent node references, we have to traverse all features in order to find the children of a feature or group. The conversion is done by building the tree recursively starting from the root. This results in quadratic complexity per time point, which yields the complexity of $O(t \cdot n^2)$.

Conclusion of Total Complexity

The analysis performed considers both the amount of time points, t, as well as the average number of features and groups per feature model, n. Analysing the individual steps of the algorithm and collecting the results gives us a worst-case time complexity of $O(t \cdot n^2 \cdot log(n))$. The average-case time complexity is slightly better, being $O(t \cdot n^2)$. It is important to note that we have kept the analysis fairly simple. We have used the average number of features and groups as well as the amount of time points as variables in the analysis. However, we have not considered the amount of modifications between each time point, which would affect the complexity of certain parts. Which types of modifications are included is also vital, where some operations are more costly than others. However, we hope that this analysis will give some indication to how the algorithm might perform.

Chapter 5

Command-Line Interface and Visualization Tools

In this chapter we go through some tools created around the three-way merge algorithm. First, in Section 5.1, we present a command-line interface wrapping the merge algorithm. Second, in Section 5.2, we introduce a tool created for exploring the input and output of the merge algorithm in a visual manner.

5.1   Command-Line Interface

The evolution plan merger defines a command-line interface, `epmerge`, which does three-way merges on evolution plans. The interface acts as a wrapper around the three-way merge algorithm, handling things like reading and writing to file, logging, converting between representations, etc.

By designing a command-line interface using the common data serialization format JSON, our application can be easily integrated with other tools. Other tools are often designed and implemented in other technologies and languages, and having a command-line interface allows for easier integration of the tools.

We used the Haskell library *optparse-applicative*[1] to define the interface, which allowed us to automatically generate a help page. By invoking the command `epmerge --help`, the following help page will be shown.

```
A three way merge tool for feature model evolution plans
```

---

[1] https://github.com/pcapriotti/optparse-applicative

```
Usage: epmerge ( --generateOne EXAMPLENAME
               | --generateAll
               | --fromFile FILENAME
               )
               [-F|--fromType FROMTYPE] [-T|--toType TOTYPE]
               [-p|--print] [-g|--generateElm]
               [-o|--toFile FILEPATH]
  Merges evolution plans into a single merged plan, which
  respects the formal semantics of evolution plans


Available options:
  --generateOne EXAMPLENAME
                        Generates one of the examples
  --generateAll         Generates all examples
  --fromFile FILENAME   Read a three way merge plan from file
  -F,--fromType FROMTYPE
                        The type to convert from (useful only
                        when reading from file) (choices:
                        TreeUser | FlatUser | FlatModification)
                        (default: TreeUser)
  -T,--toType TOTYPE    The type to convert to (useful when
                        printing and writing to file) (choices:
                        TreeUser | FlatUser | FlatModification)
                        (default: FlatModification)
  -p,--print            Whether to print the merge result
  -g,--generateElm      Whether to pass generated results
                        to the elm frontend
  -o,--toFile FILEPATH  Outputed file to write the
                        merge result as JSON
  -h,--help             Show this help text
```

Modes The interface has three different *modes*; `GenerateOne`, `GenerateAll` and `FromFile`. The first two merge either one or all of the predefined test inputs. This includes different variations of the vending machine example we will explore in this chapter. The last, mode `fromFile`, will read the input from a JSON encoded file.

- `GenerateAll` simply runs all the examples in the code. This includes some sound examples and some erroneous examples. The erroneous examples consist of `Merge` conflicts, `Local` conflicts and `Global` conflicts. This mode can be run using `epmerge --generateAll`.

- `GenerateOne` takes one argument, `Example Name`, which is the string associated with one of the examples in the code. This mode can be run using `epmerge --generateOne EXAMPLENAME`. The merger then

runs on the specified example. If you provide an `EXAMPLENAME` that does not exist, the merger will give you all the available example names.

- `FromFile` also takes an argument, `File Name`, which is the name of JSON file to read from. The merger will read the file, and run the merger on the input. This mode can be run using `epmerge --fromFile FILEPATH`.

Options The interface also defines some optional options that can specify the behavior of the merge algorithm. The different options will mainly specify the input and output formats, as well as what kind of output will be generated.

When using the `FromFile` mode, you may specify what evolution plan representation you are using. This can be done with the `--fromType` option, which takes either `TreeUser`, `FlatUser` or `FlatModification` as a parameter.

In able to view the results of the merge, we could do one or more of the following.

- Print the result of the merge using the `--print` option

- Write the result to a JSON file using the `--toFile FILEPATH` option.

- Write the example(s) to the Elm frontend using the `--generateElm` option. The next time the frontend is loaded, the user can see an actual visual, tree representation of the *base*, *version 1* and *version 2* evolution plans, as well as the expected and actual merge output. The frontend is discussed in more detail in Section 5.2.

To specify the output format of the print or the file to write the output, you can use the `--toType`, with either `TreeUser`, `FlatUser` or `FlatModification` as an argument.

As an example, running the following will read a sound example from file[2], merge the input, and print the result. `epmerge --fromFile="./data/sound_flatuser.json" --fromType=FlatUser --print`

5.2 A Visualization Tool for Evolution Plans

In this section we discuss and showcase a visualization tool for the merger we have created. This is not a tool created to be used directly by users, but

---

[2]`https://github.com/eirikhalvard/master-thesis/blob/master/backend/data/sound_flatuser.json`

rather to demonstrate the merge algorithm. Evolution plans are complex structures that are hard to grasp in textual format, especially considering the fact that we are dealing with multiple evolution plans. This tool lets us explore the input and output of the merge algorithm in a more structured, visual manner.
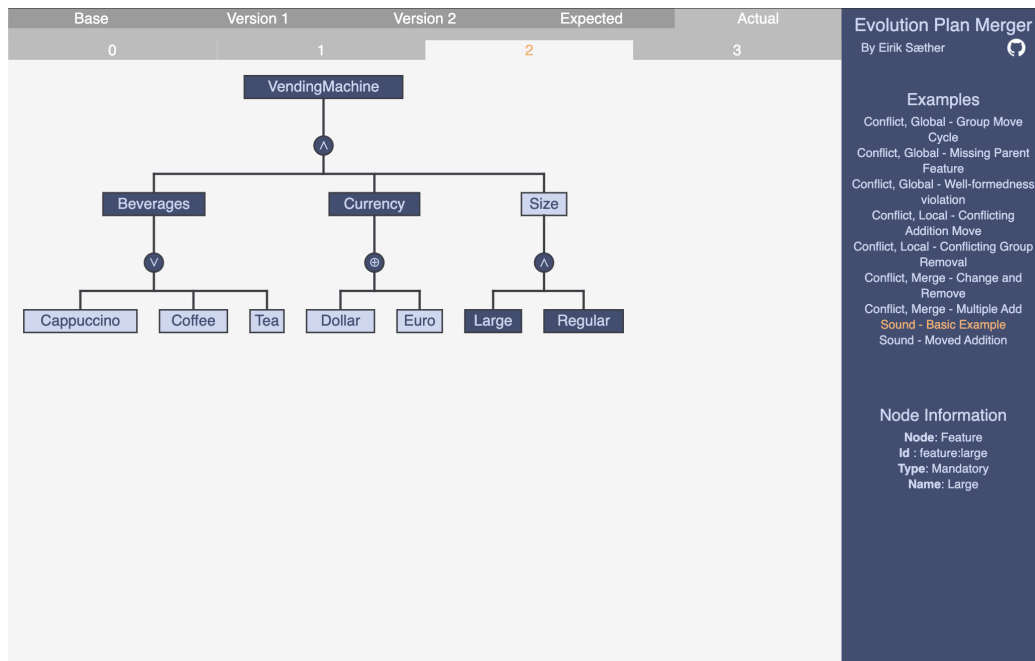
Figure 5.1: The visualization tool in use

As part of this project, we have created a multitude of different scenarios for merging. This includes examples where the individual changes are automatically unifiable without user input as well as examples that result in one or more conflicts. Every one of the predefined examples can be explored using this tool.

The tool is implemented in Elm[3], an ML language specializing in web-applications. The language we chose for the merge program, Haskell, has a very similar approach to defining types. This allows for defining types almost identical to the ones in the merger in addition to the other benefits of a purely functional ML language.

The tool is entirely separate from the command-line interface for the three-way merge algorithm. By providing certain options to the CLI, the merger can generate data that this tool can read, parse and visualize.

The visualization tool lets us choose a merge example from a predefined

---

[3]https://elm-lang.org/

list. Once an example is chosen, the tool allows us to explore the three evolution plans that are used as input to the three-way merge. In addition, the expected output of the merge algorithm as well as the actual output from the merger can also be explored.

Each evolution plan is represented as a list of feature models. By choosing a certain time point, the user is presented with a feature model visualized as a tree. Based on the information in each feature model, the tool automatically generates a tree representing the feature model. The trees are drawn using SVG[4].

To see the tool in use, see Figure 5.1. To the right, we can see the list of explorable example merges. The top section lets us pick what part of the input or output we want to see, as well as what time point we want to explore. The visualization for the trees follows a similar style to what we presented in this paper. The only difference between the tool and what we defined in Section 3.4.3 on page 29 is the representation for feature types. In this thesis, whether the feature is mandatory or optional is represented by a black or white circle above the feature. However, in the tool, this is represented by the background color of the feature. A mandatory feature type is noted by a dark blue background, whereas the optional type has a light blue background.

---

[4]`https://en.wikipedia.org/wiki/Scalable_Vector_Graphics`

Part III

Case Study and Conclusion

Chapter 6

Case Study – Vending Machine

In this chapter, we apply the three-way merge tool on examples inspired by real-world evolution plans. We do this by presenting two different examples, where one results in a successful merge, and the other results in a conflict. After presenting the examples, we discuss advantages and disadvantages of the chosen merge strategy. In the discussion, we use the presented examples to highlight the issues and benefits of the merge tool.

The examples model the evolution of a vending machine software product line. The evolution plans we present model the planned changes of common vending machine features, like beverages such as tea and coffee, different types of currency, different sizes to the cups, etc.

Since we are creating a three-way merge example, we input three distinct evolution plans to the algorithm: A base evolution plan, and two derived evolution plan, each with their own changes to the base evolution plan. The changes of both versions are then detected and merged into a final, merged version of the evolution plans.

We showcase two slightly different examples, an example that results in a sound, well-formed evolution plan, and one that results in a conflict. The examples we present are encoded, executed and tested on the tools we have created. The input to the merge tool is encoded in the common serialization format JSON, which is passed as argument to the command-line interface wrapping the three-way merge algorithm. The results from the algorithm are then serialized and written to a file. The command-line interface also generates data which is passed to the visualization tool, which visualizes the input and output of the algorithm.

6.1   A Sound Example

Firstly, we present an example of a merge that results in a sound, well-formed evolution plan. The visualizations and results we present are actual data that have been tested and verified on the merge tool. We start by presenting the three input evolution plans to the algorithm, then present the actual result of the merge tool. The example is also included in the command line interface, which can be run with the command `epmerge --generateOne SoundExample --print`.

The Base Evolution Plan

We present the base evolution plan for the sound vending machine example. This base evolution plan is represented in Figure 6.1 on the following page, which is visualized as a collection of screenshots from the visualization tool presented in Section 5.2 on page 93.

The base evolution plan consists of four distinct time points. Each time point represents a milestone in the development of the vending machine example. Each time point is associated with a feature model, where more and more components are added with each time point. In the last time point, we will see a feature model that represents all the different ways we can configure a vending machine.

In base evolution plan, we can see several different components consisting of features and groups. In time point 1, we introduce a mandatory beverages feature, which means a vending machine has to have a beverages option. Due to its or-group, a vending machine can choose if it wants to provide coffee, tea, none or both. In time point 2, a collection of features and groups are added which provides options for cup sizing of beverages. The size is optional, which means a vending machine does not have to have a size. If the size group is chosen, the vending machine has to supply both regular, large and small cups. In the last time point, 3, the milk feature is introduced. If a vending machine chooses to provide milk, it can choose to prove zero or more of the features lactose free, normal or soy milk.

Changes in Version 1

The base evolution plan introduced in Figure 6.1 on the following page represented the planned development of the vending machine software product line. However, since requirements often change, we present the base evolution plan modified with some changes to the plan. In reality, this
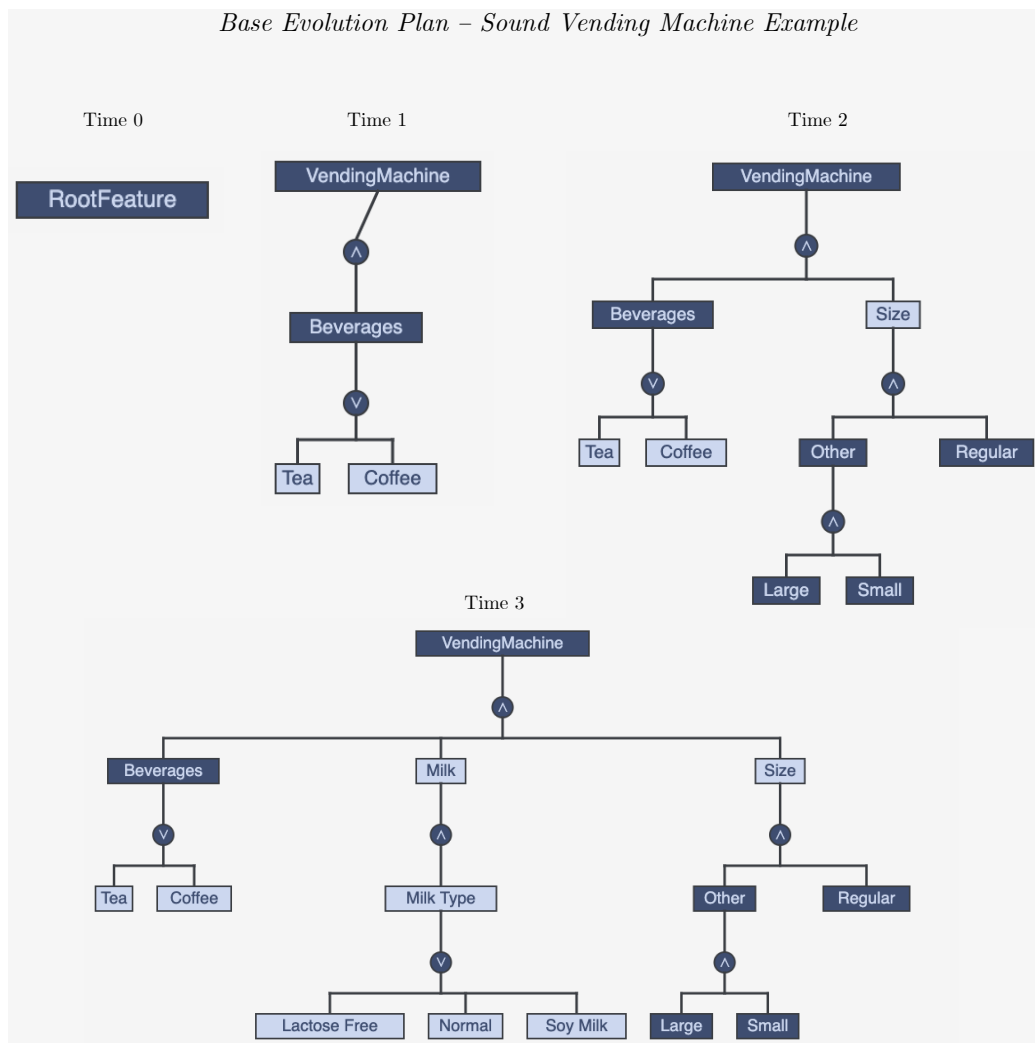
Figure 6.1: Sound Vending Machine Example - Base Evolution Plan

is encoded as a completely new evolution plan, but we will just summarize the differences between this version and the base evolution plan. The full encoding can be seen in the Github repository[1]. The changes include the following:

- At Time 1: Added a Currency feature under the Vending Machine group. This has an *alternative* group with both a Dollar and Euro feature. This indicates that a valid vending machine has to have some form of currency, which has to be either dollars or euro.

- At Time 2: Simplified the Size feature. The only two features in the and-group of size are now Regular and Large. This implies that if the vending machine allows different sized cups, the only two choices has to be regular or large.

- A Time 3: Changed the types of the milk type group and the different milk types. The Soy Milk feature has also been removed. In the base, a vending machine could choose what milk types it could have. In this version however, if the vending machine has a milk type option, both the lactose free and normal milk types have to be provided.

Changes In Version 2

We will now present the second version of the evolution plan. The changes done in this version is made on the base evolution plan. This means that the changes in version 1 has not been synchronized with this version yet. We summarize the changes in the same way as we did with version 1:

- At Time 1: An additional beverage type has been provided. We can see this by the new Cappuccino feature under the beverages group.

- A Time 3: Changed the types of the milk type group and the different milk types except the Soy Milk feature. This change is almost the same as in version 1, except that the Soy Milk feature has not been removed.

The Resulting Evolution Plan

We will now present the result of the merger. By supplying the three evolution plans, the merger will use the base evolution plan to detect what changes has been made. The changes derived has been presented above. The merger will then merge these changes, and check that the resulting

---

[1] `https://github.com/eirikhalvard/master-thesis/blob/master/backend/data/sound_treeuser.json`

evolution plan is sound. By collecting screenshots from the visualization application, we present the resulting evolution plan in Figure 6.2.
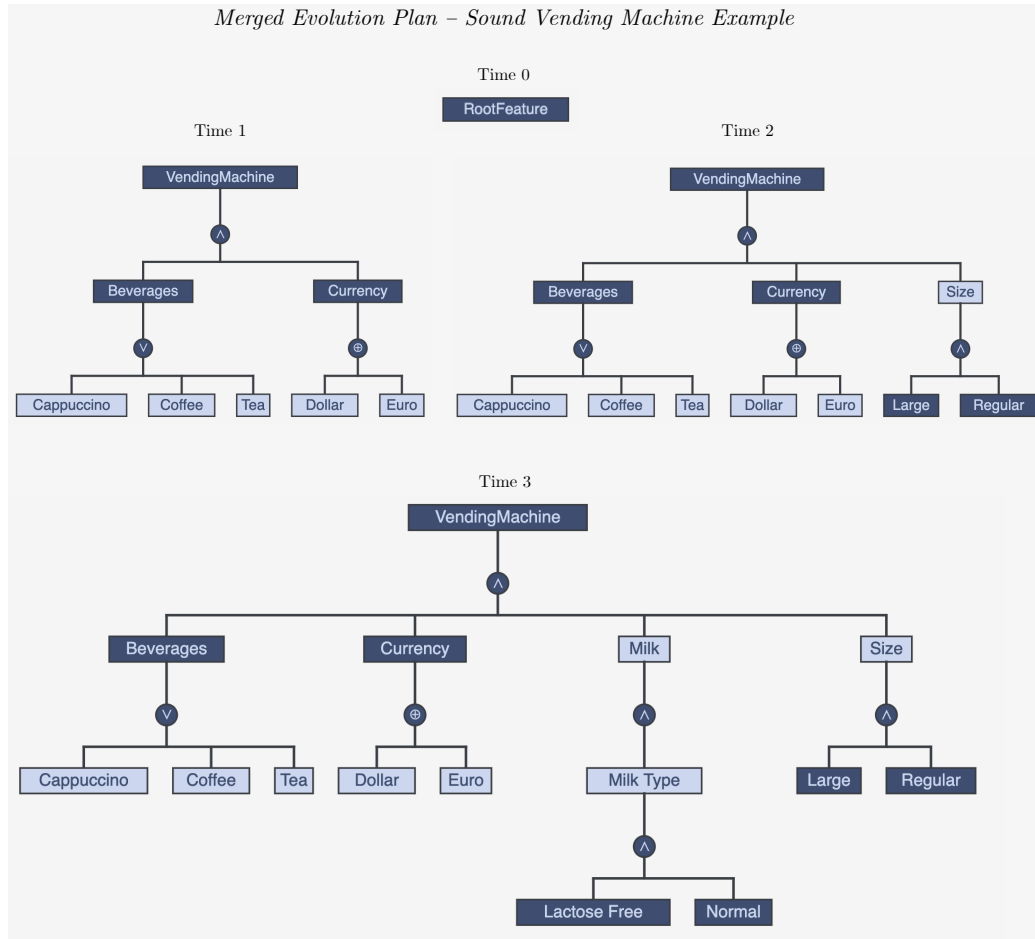
Figure 6.2: Sound Vending Machine Example - Merged Evolution Plan

The changes in version 1 and 2 are unifiable, so we get a sound evolution plan as a result. In some cases, the changes are not overlapping, which results in both changes being included. In other cases, both versions made the same change, so the change is also included in the merged evolution plan.

6.2   An Unsound Example

While the previous example resulted in a sound evolution plan, we now present an example that results in a conflict. The example we showcase is very similar to the previous example, with small changes made. The

example we showcase results in a local conflict. The conflict arises due to one version trying to add a feature at time 2 while the other version adds the same feature at time 3. The example is also included in the command line interface, which can be run with the command `epmerge --generateOne ConflictingAdditionMove --print`.

Differences from the Sound Example

The base evolution plan is exactly the same as the previous, sound example. The example can be seen in Figure 6.1 on page 99.

Version 1 is almost exactly the same as in the sound example. The main difference is when the Coffee feature is introduced. Instead of adding this feature in time 1 as in the original plan, we introduce it in time 2.

Similarly, version 2 is almost identical to version 2 in the sound example, with one exception. The addition of the Coffee feature is moved from time 1 to time 3.

Results of Merging

By executing the three way merge on the specified example, we will get a local conflict. The merger will successfully merge the modifications in time 0, 1 and 2. However, at time 3, the conflict will arise since the Coffee feature already exists.

In Figure 6.3 on the next page, we can see the actual result in the visualization application. The resulting conflict encoded in the `Local` data type is seen in below.

```
Local
  3
  (FeatureAlreadyExists
    (FeatureAdd "group:beverages-group" Optional "Coffee")
    "feature:coffee")
```

6.3   Advantages and Drawbacks of the Solution

In light of the vending machine examples discussed in this chapter, we will discuss some advantages and disadvantages of the chosen strategy for merging. We will start by discussing some problems related to detecting and fixing conflicting changes. Following these issues, we will discuss

Base   Version 1   Version 2   Expected   Actual

Error in Evolution Plan

A conflict occured at time
3

The conflict occured because the following modification could not be integrated into the merged plan
Feature Addition: parent = "group:beverages-group", type = Optional, name = "Coffee"
on feature id feature:coffee
because the feature already exists!

Evolution Plan Merger
By Eirik Sæther

Examples
Conflict, Global - Group Move Cycle
Conflict, Global - Missing Parent Feature
Conflict, Global - Well-formedness violation
Conflict, Local - Conflicting Addition Move
Conflict, Local - Conflicting Group Removal
Conflict, Merge - Change and Remove
Conflict, Merge - Multiple Add
Sound - Basic Example
Sound - Moved Addition

Node Information
Node: -
Id : -
Type: -
Name: -

Figure 6.3: Result of merging the erroneous example

some advantages of the merge strategy, and how the advantages might make the issues manageable.

Little Detection of Cause The merge tool we have created is able to detect paradoxes in the merged evolution plan, but not what caused the paradox. When the merge algorithm is instantiated, we assume that each inputed evolution plan is sound. If a conflict occurs in the merged plan, it will occur due to incompatible changes in the two derived versions. This means that if one version has scheduled a change in the evolution plan, this change could cause issues in the planned changes for the other version. If we include the changes of one version, then arrive at the other version's incompatible changes, the merger only reports why the change will result in an invalid evolution plan. Figuring out the cause is left up to the user to do manually, as the merger will not report the underlying issue which is that the two changes were incompatible.

We witness an example of this phenomenon in the unsound vending machine example described in Section 6.2 on page 101. The original plan had scheduled a feature addition in time 1, but both versions moved the addition to time 2 and time 3 respectively. When the merger arrived at time 2, the addition from version 1 was included. However, when the merger arrived at time 3, the feature addition was not possible to perform since the feature already existed. As we can see in Figure 6.3, the only information about the merge failure was which change caused the halt and why the change could not be included. However, we did not get information about the change that caused the issue, which was that feature

was added in time point 2.

Bias Towards Earlier Changes Since we are merging time points incrementally, errors usually occur later in the plan. When we have conflicting changes in separate time points, the conflict does not arise at the change earlier in the plan, but rather at the later change. This causes a disproportionate amount of conflicts to report a problem with later changes rather than early changes. This bias towards earlier changes may cause the user to think that the issue is primarily with the later change. However, the issue may be either one, and solving the conflict requires further inspection by the engineer.

Revisiting our unsound example from Section 6.2 on page 101, we can imagine this being a potential issue. In each version, we moved a feature addition to time point 2 and 3 respectively, which resulted in a conflict in the last time point. The conflict informs that the last change in time point is problematic since the feature already exist. A simple solution to handle the conflict is to just remove the feature addition in time point 3. However, neither adding the feature at time 2 og 3 is correct, and removing the last feature addition might not be the correct solution.

Transparent Merging As we now have discussed, the choice of merging strategy raised some potential issues with conflicts. However, the strategy might come with some advantages as well. The three-way merge algorithm is designed as an incremental merger. For our case, this means that we will go through each time point and incrementally build a merged feature model at each time point. The engineer performing the merge can understand how the evolution plans are analyzed and merged, as there are no advanced analyzing across time points. The choice of a straight forward merging strategy can be perceived as more transparent than other more advanced strategies. This transparency might make it easier to solve conflicts.

Finding the cause of the conflict might be easier since the user can manually simulate how the merger operates. Since the engineer knows why the modification violated the semantics, the engineer could go back and find the change that caused it. Due to the mergers incremental nature, finding the cause requires only inspecting the same time point or any of the time points prior to the conflicts time point. By manually stepping through each time point, the cause might be determined.

Chapter 7

Conclusion and Future Work

In this concluding chapter, we start by describing how the solution addresses the research questions stated in the introduction. Further on, we describe potential future work in extending the solution to create better results. Lastly, we conclude the thesis by summarizing its contributions.

7.1   Addressing Research Questions

We start by addressing the research questions detailed in Section 1.4 on page 4.

RQ1: How do we design an algorithm for merging evolution plans?

When we designed the merge algorithm for evolution plans, we opted for a three-way approach. Doing so allows us to more accurately derive what changes each version has made to the evolution plan, which we leverage in the merge algorithm. In creating the algorithm, we split the task in several distinct parts, which we outline in Section 3.3 on page 23. The first phases include transformations of the evolution plans with representations better suited for detecting changes and merging the versions. Later steps include merging the actual changes and lastly ensuring soundness of the result. Since we have defined a precise, mathematical definition of evolution plans, we are able to leverage the properties to create a merger that respects both structure and semantics.

RQ2: How do we create a merge tool for evolution plans that respects soundness?

As evolution planning is a delicate matter, requiring attention to detail, we wanted the tool to guarantee a result that upholds the soundness requirements of evolution plans. We have managed to guarantee soundness in a couple of ways. Firstly, we assume that the evolution plans input for merging are sound. The existing tools already have methods for ensuring soundness for a single plan, and there is no point in merging multiple plans if the individual plans are unsound to begin with. This means that we can leverage the soundness of the individual plans in the merger. Secondly, we have created representations for evolution plans that allow us to guarantee certain aspects of the merging. For instance, we know that a certain feature was only either modified, added or removed once per time point, which we could not guarantee if we used other representations. Upon a successful merge, the representation allows us to guarantee soundness no matter what combination of changes both versions have made. The representation also allows us to not worry about what order in which we apply the changes per time point. This means that we can apply every change in a time point, then later go through the result and make sure there was no violations to soundness.

RQ3: How can we make predictable and interpretable results of the merge tool?

An important aspect to the merge tool is that actual engineers have to interpret and act on the results of the merge. For this reason, we wanted the results of the merge to make sense from the engineers' point of view. We specifically want the results to be predictable, which means that the merger does not make any big decisions altering the semantics of the resulting evolution plan. Predictability also means that the way the evolution plans are merged is possible to understand and predict based on the changes in each version. This is important in order to get results that reflect the intended evolution of the SPL. Upon a conflicting result, we wanted the reported reason for conflict to be possible to interpret such that changes to the evolution plan could be made accordingly. For this reason, we have opted for a merging strategy that merges one time point at a time incrementally. We felt that this approach also reflects how the engineer might have done the task if a manual approach was necessary. Upon failure, the tool provides information detailing exactly why a certain change was not possible to include. This allows the engineer to manually inspect the evolution plans to make alterations.

7.2 Future Work

For possibilities of future work, we propose the following:

- **Cause detection for conflicts**: When a change to the evolution plan would cause a violation of soundness, the merger raises a conflict. The conflict occurs because the change is not possible to implement in the merged plan. However, the *cause* of the paradox is usually due to a conflicting change in the other evolution plan version. For instance, if one version tries to remove a feature, then another version tries to add a group to the feature, we will get a conflict. The conflict will tell us that the feature does not exist, so we cannot add a group to it. However, we will not be notified that we removed the feature in the other version, which caused the paradox. Implementing functionality for cause detection might help the user understand the errors and how to fix them.

- **Constructing a version control system**: While the merge tool is an important piece in facilitating synchronization of contributions, it should optimally be complimented by a version control system. Such a system could keep record of different snapshots and branches of the evolution plan. This would allow for implementing systems for merging two branches, which could automatically fetch the common base evolution plan and pass it along to the three-way merger.

- **Integration with current tools**: A possible version control system could also be tightly integrated in the DarwinSPL editor for a more seamless experience. Currently, the DarwinSPL tool allows for constructing and replanning of evolution plans by using a graphical user interface. The merge tool or a potential version control system could be integrated in the graphical editor as well. This opens up possibilities for a more automatic synchronization process than we would using them as a standalone tools.

7.3 Conclusion

In this thesis, we have created a merge tool for feature model evolution plans. The merge tool is a three-way merger that merges two different versions with respect to the common evolution plan they were derived from. Being a syntactic and semantic merger, the merger will consider the structure and semantics of the evolution plans being merged. By not creating a general merge tool, but rather a tool specifically made for evolution plans, we leverage the specific semantics of evolution

plans to create accurate merge results. The implementation relies on a strongly typed functional language, which allows for stronger correctness guarantees than less strict languages would.

We have also created an application facilitating further development and integration with other tools. The tool has been implemented in a Haskell application with a command-line interface. Haskell being a compiled, industry-level language allows for further integration with existing tools, leveraging the defined command-line interface. A visualization tool has also been created which allows exploration of how the merge tool operates and handles different cases, which could aid further development and integration of the merge tool.

To conclude, we believe that the merge tool we created can give engineers a better toolbox for dealing with parallel development of evolution plans. The merge tool can synchronize multiple versions of evolution plans while simultaneously ensuring correctness for the evolution plan semantics, which we believe current tools are not capable of.

Bibliography

[1]   P. Clements and L. M. Northrop, *Software product lines - practices and patterns*, ser. SEI series in software engineering. Addison-Wesley, 2002, ISBN: 978-0-201-70332-0.

[2]   K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005, ISBN: 978-3-540-24372-4. DOI: 10.1007/3-540-28901-1. [Online]. Available: https://doi.org/10.1007/3-540-28901-1.

[3]   I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela, "Software diversity: State of the art and perspectives", *Int. J. Softw. Tools Technol. Transf.*, vol. 14, no. 5, pp. 477–495, 2012. DOI: 10.1007/s10009-012-0253-y. [Online]. Available: https://doi.org/10.1007/s10009-012-0253-y.

[4]   D. S. Batory, "Feature models, grammars, and propositional formulas", in *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, J. H. Obbink and K. Pohl, Eds., ser. Lecture Notes in Computer Science, vol. 3714, Springer, 2005, pp. 7–20. DOI: 10.1007/11554844_3. [Online]. Available: https://doi.org/10.1007/11554844%5C_3.

[5]   J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr, "Reasoning about product-line evolution using complex feature model differences", in *Software Engineering 2017, Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-24. Februar 2017, Hannover, Deutschland*, J. Jürjens and K. Schneider, Eds., ser. LNI, vol. P-267, GI, 2017, pp. 67–68. [Online]. Available: https://dl.gi.de/20.500.12116/1274.

[6]   G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski, "Evofm: Feature-driven planning of product-line evolution", in *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering, PLEASE 2010, Cape Town, South Africa, May 2, 2010*, J. Rubin, G. Botterweck, M. Mezini, I. Maman, and A. Pleuss, Eds., ACM, 2010, pp. 24–31. DOI: 10.1145/1808937.1808941. [Online]. Available: https://doi.org/10.1145/1808937.1808941.

[7] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. J. P. de Lucena, "Refactoring product lines", in *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, Eds., ACM, 2006, pp. 201–210. DOI: 10.1145/1173706.1173737. [Online]. Available: https://doi.org/10.1145/1173706.1173737.

[8] A. Hoff, M. Nieke, C. Seidl, E. H. Sæther, I. S. Motzfeldt, C. C. Din, I. C. Yu, and I. Schaefer, "Consistency-preserving evolution planning on feature models", in *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, R. E. Lopez-Herrejon, Ed., ACM, 2020, 8:1–8:12. DOI: 10.1145/3382025.3414964. [Online]. Available: https://doi.org/10.1145/3382025.3414964.

[9] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines", in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, A. Cleve, F. Ricca, and M. Cerioli, Eds., IEEE Computer Society, 2013, pp. 25–34. DOI: 10.1109/CSMR.2013.13. [Online]. Available: https://doi.org/10.1109/CSMR.2013.13.

[10] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines", in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds., ACM, 2008, pp. 311–320. DOI: 10.1145/1368088.1368131. [Online]. Available: https://doi.org/10.1145/1368088.1368131.

[11] C. Seidl, I. Schaefer, and U. Aßmann, "Deltaecore - A model-based delta language generation framework", in *Modellierung 2014, 19.-21. März 2014, Wien, Österreich*, H. Fill, D. Karagiannis, and U. Reimer, Eds., ser. LNI, vol. P-225, GI, 2014, pp. 81–96. [Online]. Available: https://dl.gi.de/20.500.12116/17067.

[12] M. Nieke, G. Engel, and C. Seidl, "Darwinspl: An integrated tool suite for modeling evolving context-aware software product lines", in *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2017, Eindhoven, Netherlands, February 1-3, 2017*, M. H. ter Beek, N. Siegmund, and I. Schaefer, Eds., ACM, 2017, pp. 92–99. DOI: 10.1145/3023956.3023962. [Online]. Available: https://doi.org/10.1145/3023956.3023962.

[13] R. Mitschke and M. Eichberg, "Supporting the evolution of software product lines", in *ECMDA Traceability Workshop*, 2008. [Online]. Available: http://tubiblio.ulb.tu-darmstadt.de/34815/.

[14] D. Hinterreiter, H. Prähofer, L. Linsbauer, P. Grünbacher, F. Reisinger, and A. Egyed, "Feature-oriented evolution of automation software systems in industrial software ecosystems", in *23rd IEEE*

International Conference on Emerging Technologies and Factory Automation, ETFA 2018, Torino, Italy, September 4-7, 2018, IEEE, 2018, pp. 107–114. DOI: `10.1109/ETFA.2018.8502557`. [Online]. Available: `https://doi.org/10.1109/ETFA.2018.8502557`.

[15] F. Schwägerl, "Version control and product lines in model-driven software engineering", PhD thesis, University of Bayreuth, Germany, 2018. [Online]. Available: `https://epub.uni-bayreuth.de/3554/`.

[16] W. 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*, J. F. H. Winkler, Ed., ser. Berichte des German Chapter of the ACM, vol. 30, Teubner, 1988, pp. 1–20.

[17] T. Mens, "A state-of-the-art survey on software merging", *IEEE Trans. Software Eng.*, vol. 28, no. 5, pp. 449–462, 2002. DOI: `10.1109/TSE.2002.1000449`. [Online]. Available: `https://doi.org/10.1109/TSE.2002.1000449`.

[18] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest subsequences", *Commun. ACM*, vol. 20, no. 5, pp. 350–353, 1977. DOI: `10.1145/359581.359603`. [Online]. Available: `https://doi.org/10.1145/359581.359603`.

[19] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel changes in large-scale software development: An observational case study", *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 3, pp. 308–337, 2001. DOI: `10.1145/383876.383878`. [Online]. Available: `https://doi.org/10.1145/383876.383878`.

[20] J. Buffenbarger, "Syntactic software merging", in *Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Selected Papers*, J. Estublier, Ed., ser. Lecture Notes in Computer Science, vol. 1005, Springer, 1995, pp. 153–172. DOI: `10.1007/3-540-60578-9\_14`. [Online]. Available: `https://doi.org/10.1007/3-540-60578-9%5C_14`.