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

Three-Way Semantic Merge for Feature Model Evolution Plans

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Acknowledgements

[[TODO: write]]

Abstract

A software product line (SPL) models closely related software systems by capitalizing on their similarities and differences. Software engineers explicitly encode similarities and differences of an SPL by defining it in terms of a feature model. However, 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 not done by a single engineer, but multiple engineers, often working separately and independently of each other. To ensure successful 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

Preface

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

something about summer project?

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 in the SPL have several things in common, as well aspects that separate them. As an software development paradigm, SPLs allows the developers to leverage the softwares commonalities and variabilities in order to improve a variety of factors.

This variability and commonalities can be captured by *feature models (FM)*, which uses *features* to capture the different aspects of an SPL. 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 [1]. The feature model organizes the features in a tree structure to capture the variability of the SPL.

Since SPLs undergo continuous evolution, planning for the long-term evolution of a software product line is often crucial. The evolution of the SPL is mostly 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.

We address the lack of tools for long-term evolution planning with *feature model evolution plans*, or just *evolution plans*, which not only models the current feature model, but all intended future feature models. 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.

Planning the evolution of the software product line often involves multiple engineers changing and evolving the plan. Therefore, creating tools for evolution planning requires synchronization techniques for allowing collaborators to work independently. Naively integrating each persons changes to the evolution plan may yield inconsistencies and conflicts, so we investigate different merging strategies to ensure a sound, well-formed evolution plan is produced after the merge.

[[TODO: scope: her modellerer jeg bare planlegging.]] [[TODO: hvordan min oppgave skal leses.]]

1.1 Motivation

Evolution plans are designed to help engineers cope with the long-term evolution of software. Designing the software is an iterative and dynamic process, which is subject to change. Having several engineers working in parallel on the same evolution plan can be beneficial in handling the dynamic nature of evolution planning. This requires good synchronization tools for handling several engineers working and changing the evolution plan.

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, all following the formally defined structure and semantics. The algorithm will then try to merge the two derived plans, with the base model as a reference to what changes were made. Any conflict that occur when merging will be dealt with by consulting the user and providing options for what to do. When all conflicts are dealt with, a new merged evolution plan is produced, where the algorithm ensures that the merged plan follows the strict structure and semantics defined. This task is not trivial, and 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.

1.3 About the Project

ltep stuff blabla

1.4 Research Questions

As the goal of the thesis is to provide tools aiding developers and engineers cooperate in planning the evolution of a software product line, we propose some research questions (RQ). The research questions are concrete questions that we want to answer as a result of this thesis:

RQ1 *Is it feasable to create a merge tool for evolution plans that respect soundness?* When developing a merge tool, we optimally want to ensure that the structure and semantics of evolution plans are met. Can the tool ensure this?

RQ2 *Are the results of the merge tool predictable and interpretable?* As we want the merge tool to be used by humans, the output of the tool needs to be possible to understand.

RQ3 *What are the drawbacks and short comings of the merge tool?* The method of merging may have several implications for the result of the merge. We want to investigate if there are any issues as a result.

1.5 Contributions

The main contribution of this thesis is a three-way merge algorithm for feature model evolution plans that ensures a well-formed, sound plan 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 are 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 file, logging the output of the algorithm, checking for correct behaviour, 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 behaviour of the program matches the actual output.

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 tools for exploring the results of the merge in Elm
- Provided test cases and examples that results in both sound evolution plans and merge conflicts

1.6 Chapter Overview

Chapter 2 something about background

Chapter ?? something about something

[[TODO: WRITE]]

1.7 Project Source Code

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

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

Chapter 2

Background

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

[[TODO: begynne med å introdusere background. fortelle hva vi skal gå gjennom. fortelle om spl. div merge teknikker. haskell formalisering. etc]]

2.1 Software product lines

A software product line (SPL) is a family of closely related software systems. These systems will often have several features in common, as well as variations that makes each piece of software unique. SPLs are used to make highly configurable systems, where each product in the SPL, called a *variant*, is defined by the combination of features chosen. Several large scale companies such as Hewlett-Packard and Nokia are finding remarkable quantitative improvements in product quality, customer satisfaction, and more by using a software product line approach [5].

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

2.1.1 Feature Models

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

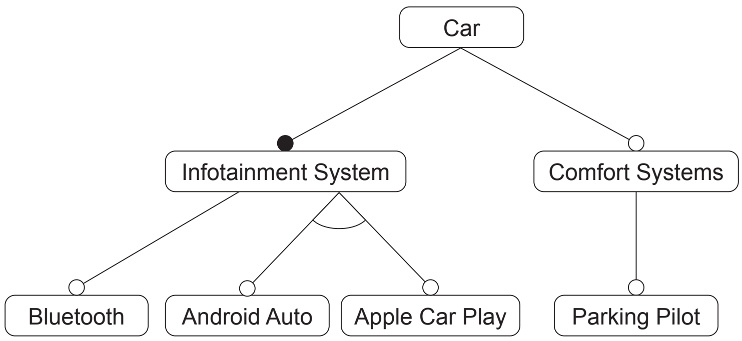


Figure 2.1: Example feature model

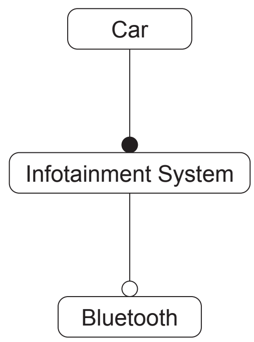
A visual representation of a feature model can be seen in Figure 2.1. The small dot above `Infotainment System` indicates that the feature is mandatory, where as the white dot above `Comfort Systems` represents an optional feature. Each feature (except the root) is in a group. The `Infotainment System`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

2.1.2 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. Planning for the long term evolution of the product line can be important in managing the complexity that comes with large software systems. Developing these kinds of systems typically involves many engineers, managers or other stakeholders, and managing when certain changes, additions or deprecations are implemented can be complex and confusing without suitable tools. Changing the SPL potentially influences many configurations, which might conflict with the stakeholders requirements.

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

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



At time 1:

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

At time 2:

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

Figure 2.2: An example evolution plan

An example of an evolution plan can be seen in Figure 2.2. The initial feature model contains three features, and two time points are added. At time 1, a group and two features are added, and at time 2, another group and two features are added. The evolution plan can derive three feature models, the initial, and the two at time 1 and 2. Performing all the operations results in a feature model that is equal to the one in Figure 2.1 on page 7

2.2 Version Control Systems

Software configuration mechanisms is the discipline of managing the evolution of large and complex software systems [8]. *Version control mechanisms* are used to deal with the evolution of software products. These mechanisms include ways to deal with having multiple, parallel versions of the software simultaneously. Techniques like *software merging* are used to keep consistency and unify different versions by automatically or semi-automatically deriving merged versions of the different parallel versions.

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

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 merge will know extra information from the common ancestor.

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

2.2.2 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 computes the difference between files by comparing equality over the lines. This has several implications, like adding a single space after a line is considered a deletion of the old line and addition of the new. This coarse granularity often leads to unnecessary and confusing conflicts. Changing the indentation or other formatting differences often lead to unnecessary conflicts.

To exemplify this, consider the two versions of a Python program, Listing 1 and Listing 2 on the following page. The second version simply wrapped the content of the function in an if-statement that checks for input sanity. Using a standard textual, line-based differencing tool like the Unix' *diff*-tool [3], we are able to calculate the difference between the two files by calculating the longest common subsequence. As seen in the result (Listing 3 on the next page), difference between the two are confusing and inaccurate. Conceptually, the difference is that the second version wrapped the block in a if-statement. Due to the coarse grained line-based differencing and the disregard of structure and semantics, the algorithm reports that the whole block is deleted, and the same block wrapped in an if is inserted.

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

Listing 1: Code diff 1

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

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

Listing 2: Code diff 2

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

Listing 3: Resulting code diff

changed files could be merged automatically [6]. Other tools can complement the merge algorithm in avoiding or resolving conflicts. Formatters can make sure things like indentation and whitespace are uniformly handled, to avoid unnecessary conflicts. Compilers can help in resolving conflicts arising from things like renaming, where one version renames a variables, while another version introduces new lines referencing the old variable.

2.2.3 Syntactic Merging

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

The granularity of the merger is still relevant, because we sometimes want to report a conflict even though the versions can be automatically merged. Consider the following example. $n < x$ is changed to $n \leq x$ in one version, and to $n < x + 1$ in another. Too fine grained granularity may cause this to be merged conflict free as $n \leq x + 1$. The merge can be done automatically and conflict free, but here we want to report a warning or conflict, because the merge might lead to logical errors.

2.2.4 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 a constraint is references to a variable. The variable references in the code are often semantically tied to the definition of the variable, where the name and scope implicitly notes the cross tree reference to the definition.

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

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

2.3 Haskell and Algebraic Data Types

[[TODO: write about type synonyms, data types including records and sum types. show polymorphic data types. maybe something about deriving, maybe lens generation? etc. something about maybe? expressive and strict type system]]

Part II

A Soundness-Preserving Merger for Evolution Plans

Chapter 3

Developing a Sound Three-Way Merge of Evolution Plans

In this chapter, we want to define the general characteristics of our tool for merging evolution plans. We will define precisely what we are trying to merge, and how we are planning to do it. We will discuss the context of the tool, as well as how it could be integrated with other tooling. We will 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 will also be presented, due to its importance in creating a merger producing sound plans.

3.1 Problem Definition

Evolution planning is a long term endeavour, often requiring planning years a head when dealing with large scale SPLs. Software product lines are subject to frequent changes, which requires 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 are a created 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 will take 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 will merge version 1 and version 2, using the base version to create better results. The merged 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 9, 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 11 and 2.2.4 on page 12

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 core platform engineers will use, is the Eclipse application Darwin-SPL[1]. In this 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.

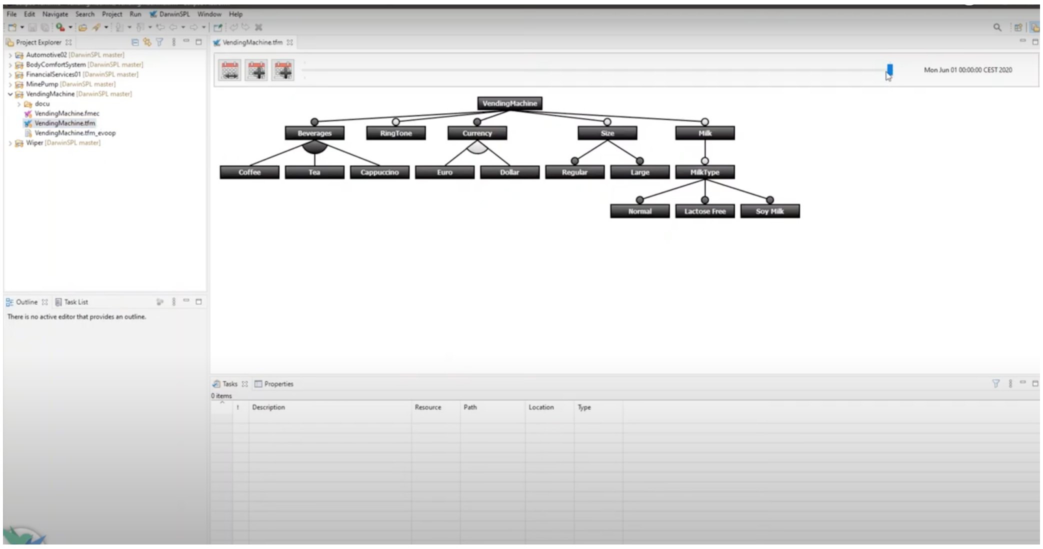


Figure 3.1: DarwinSPL Screenshot

As the merger is designed with the DarwinSPL application in mind, the merge tool is not intended to be interacted with directly. The tool I 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.

Our three-way merge backend is implemented in Haskell, which is different from the DarwinSPL tool which is implemented in the Java language. For this reason, one could not simply plug the pure three-way merge algorithm directly. To tackle this problem, we have defined

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

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 82

3.3 Algorithm Overview

In this section we will 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 29

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 45

Unifying Changes The information from the previous phase will be used to create a single merged evolution plan. This evolution plan is simply just the *base* evolution plan integrated with all the changes from

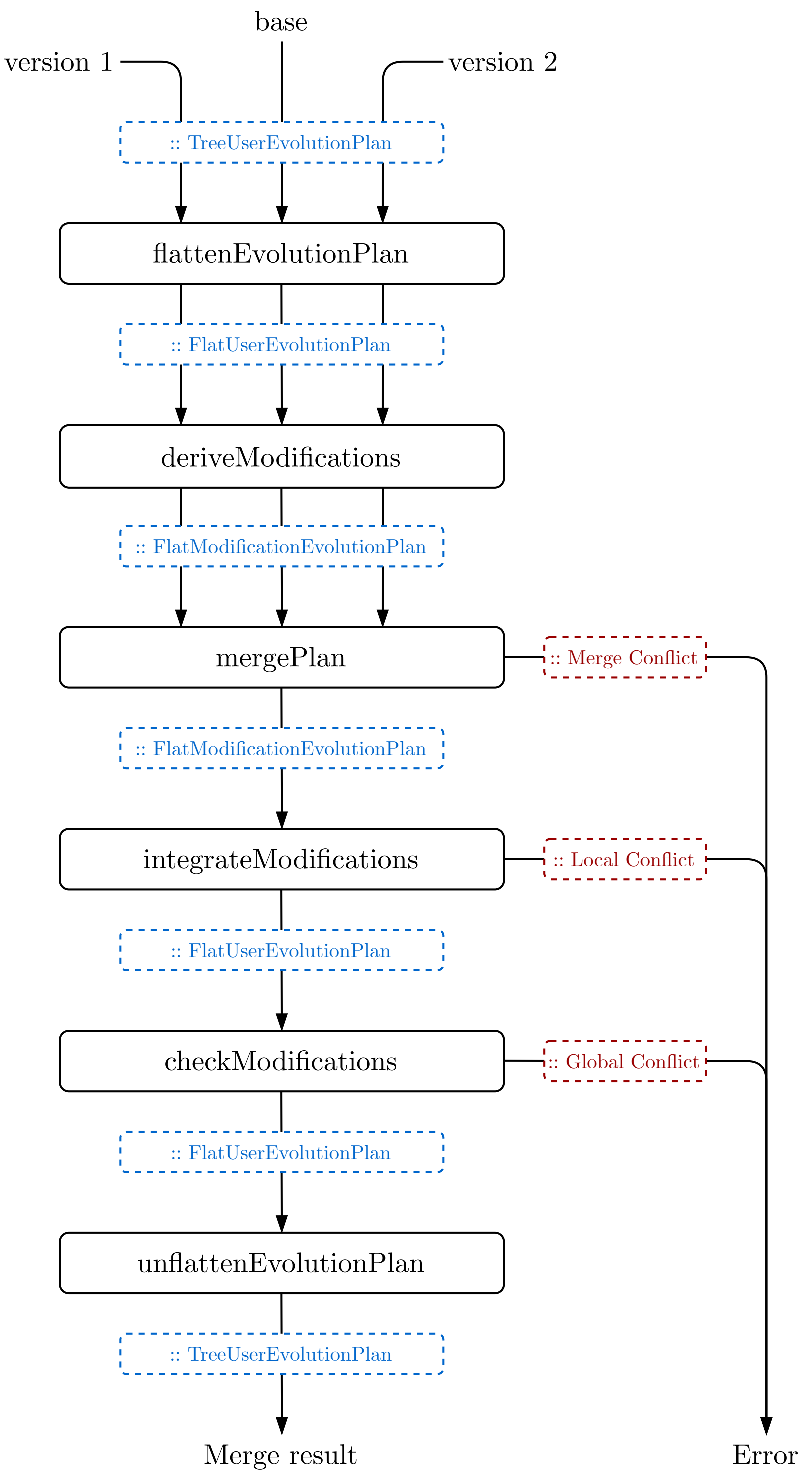


Figure 3.2: Outline of the three-way merge algorithm

version 1 and *version 2*. This phase is part of the `mergePlan` function, which is described in further detail in 4.4 on page 56

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 61

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

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, were we have the three representations `TreeUserEvolutionPlan`, `FlatUserEvolutionPlan` and `FlatModificationEvolutionPlan`. We will go into further detail about the different representations and their purpose later, but we give a small 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 17. 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 first, 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 will make the process of the three-way merge significantly easier.

3.4 Formal Definition of Evolution Plans

As stated in **RQ1** in Section 1.4 on page 4, we want 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 [7] 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 [7], there was 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 in this thesis, I will build upon the semantics of the operations defined in [7]. However, we cannot use the operations directly, since due to several issues we will detail in Section 4.2.2 on page 33. 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.

In the diagram, each feature is a node of rectangular shape with the name of the feature in the center of the node. Features has either a black or white circle above it, indicating whether the feature is mandatory or optional. This is true for every feature but one, 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 are using the $\wedge$ symbol, *or* groups use $\vee$, and *alternative* groups use the $\oplus$ symbol.

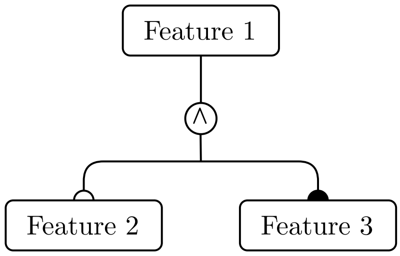


Figure 3.3: Visual representation of an example feature model

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

4.1 A Normal Form for Evolution Plans

Before diving into the inner workings of the three-way merge algorithm, we will 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 will 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 will look at the tree representation for feature models, formalized as `TreeFeatureModel`.

```haskell
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 is a few important things to note in this representation. Each feature and groups id is unique across the entire feature model, which we will leverage when merging. Each feature and group can have an arbitrary number of children. The children is organized in a `Set`, not a `List`, noting that the ordering is irrelevant. In order for the feature model to be sound, some combination of parent group type 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 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 defining it twice.

```
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, `TreeUserEvolutionPlan`. This is relatively straight forward, the only thing we have to do is replace the `featureModel` argument with our concrete feature model, namely `TreeFeatureModel`.

```
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 29, we will show how this example are transformed in the various evolution plan representations. In Section 4.3 on page 45, 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.

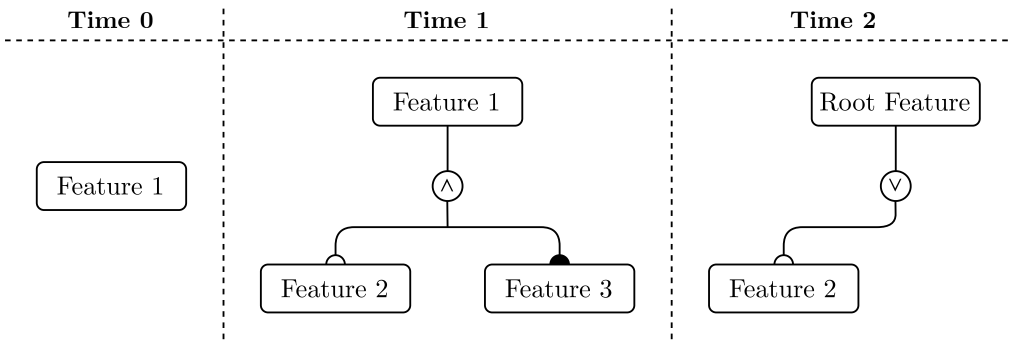


Figure 4.1: A Simple Evolution Plan Example

```
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 are 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 25, `TreeUserEvolutionPlan`, 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 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. In such events, the additions, deletions and modifications to features and groups has 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 36 go into more detail on formal definitions, algorithms and examples.

A Flatter 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 difference 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 straight forward, 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 Flatter 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 has advantages will make deriving and integrating modifications between two subsequent feature models easier. Since we have a flat structure, deriving or integrating modifications on a feature or group saves us from traversing the tree 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 don't need to modify the parent node when removing a node. This makes moving entire subtrees straight forward, requiring changing only the parent-field of the node to move.

Formalizing the Flat Structure

We start by defining the new representation for feature models.

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

data FlatFeature = FlatFeature
  { parentGroupId :: Maybe GroupId
  , featureType :: FeatureType
  , name :: String
  }
```

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

The definitions of `FeatureType`, `FeatureId`, `GroupType` and `GroupId` is still the same as defined in Section 4.1.1 on page 25.

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, `UserEvolutionPlan`, we can instantiate it in the following way.

```haskell
type FlatUserEvolutionPlan = UserEvolutionPlan FlatFeatureModel
```

Continuing the Simple Example

With this new representation for evolution plans, the example from Figure 4.1 on page 27 can be encoded in the following way:

```haskell
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"
        [
          ( "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 is 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 is sound. In order to do this transformation, we would need to go through every feature model in the plan and transform it using `flattenSoundFeatureModel`

```
flattenSoundEvolutionPlan ::
  TreeUserEvolutionPlan ->
  FlatUserEvolutionPlan
flattenSoundEvolutionPlan =
  L.timePoints
    . traversed
```

```
  . L.featureModel
  %~ flattenSoundFeatureModel
```

Flattening a feature model is relatively straight forward, 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`.

```
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 [7], 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 36 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 editing we could perform on features and groups, like `addFeature`, `removeGroup`, `moveFeature` and `renameFeature`. The effects and conditions of the operations were also formally defined in the paper. These operations were organized in a list, implying that the ordering are important. By applying the operations one by one on the current feature model, we would achieve 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 sees are a list of feature models associated with time points. Exactly how the evolution plan was manipulated, and in what order each operation took place is not relevant for the user, so it should also not be relevant for the merge algorithm.

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` → `G1` → `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` → `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 4.2.2.

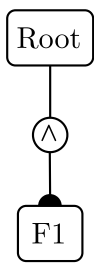
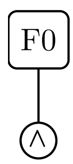
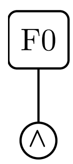
Initial Model	Operations	Original Order	Swapped Order
	Rename(Root, F0) Remove(F1)		

Table 4.1: Non-Dependent Operations Example

This meant 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 don't 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 the operations still matter, 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 will demonstrate an example were 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 4.2.2.

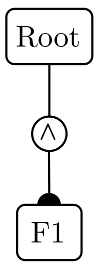
Initial Model	Operations	Original Order	Swapped Order
	Remove(F1) Remove(G1)	Root	Conflict: Removal of non-empty group

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 operation 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` $\rightarrow$ `G1` $\rightarrow$ `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` $\rightarrow$ `G1` $\rightarrow$ `Feature 1`, are 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` $\rightarrow$ `G1` $\rightarrow$ `Feat 1`.

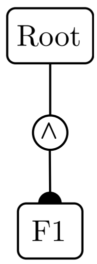
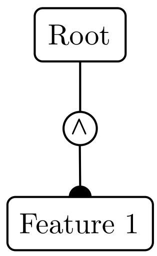
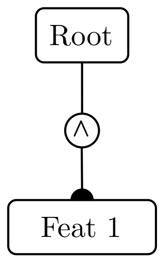
Initial Model	Operations	Original Order	Swapped Order
	Rename(F1, Feat 1) Rename(F1, Feature 1)		

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, `FlatModificationEvolutionPlan`, 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 consists of two mappings, one for features and one for groups. Each mappings map ids to a *modification*. By creating a indexed mapping structure, we limit a feature or group to have maximum one modification at a time point. This could be either an addition, removal or a change to one or more fields.

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

The new semantic for detecting changes between feature models has an important implication. The operation-based semantic allowed 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 `FlatModificationEvolutionPlan` 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 will reuse 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 FeatureNameModification
  = FeatureNameModification String

data FeatureTypeModification
  = FeatureTypeModification FeatureType

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 26. 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))
  )
]
```

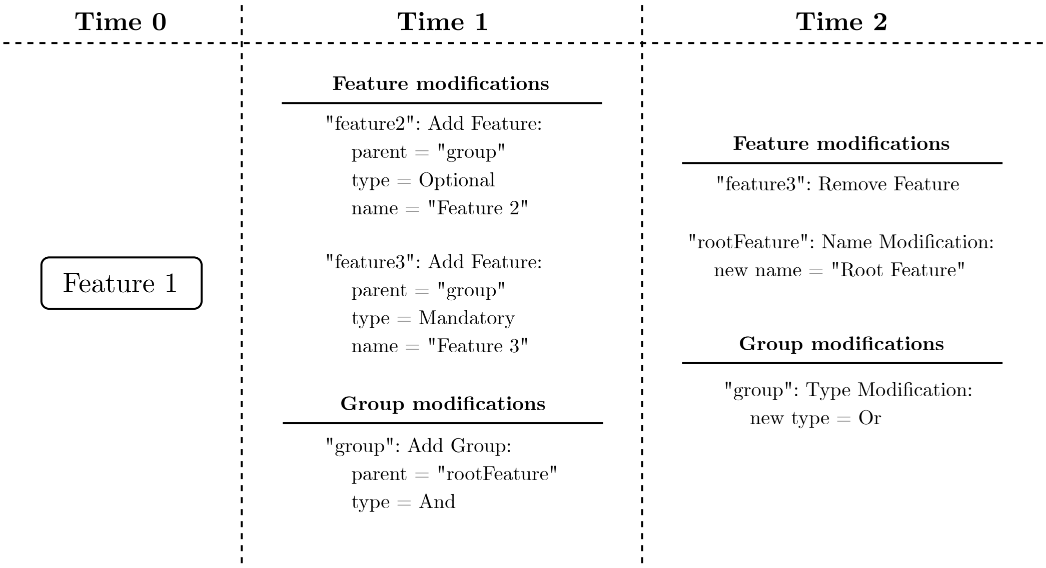


Figure 4.2: A Simple Evolution Plan Example

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 will use 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 have assumed soundness for our plans, which is why the actual function in the code is called `deriveSoundModifications` and not deriveModifications 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 will be done using the `merge` function from `Data.Map.Merge`, which will compare the keys, the feature ids, in both maps. Based on the result of the comparison, one of three different merge tactics will be used to produce the desired result. The merge tactics will rely on the three functions that we pass to the `merge` function. Given a feature id, the following three cases describes 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 equal, 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.

```haskell
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
       in if prev == new
            then Nothing
```

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

```
            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.

```
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 will be passed to the `diffFeatureModels` function, which will derive the modifications. As described in the code above, the function will compare 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.

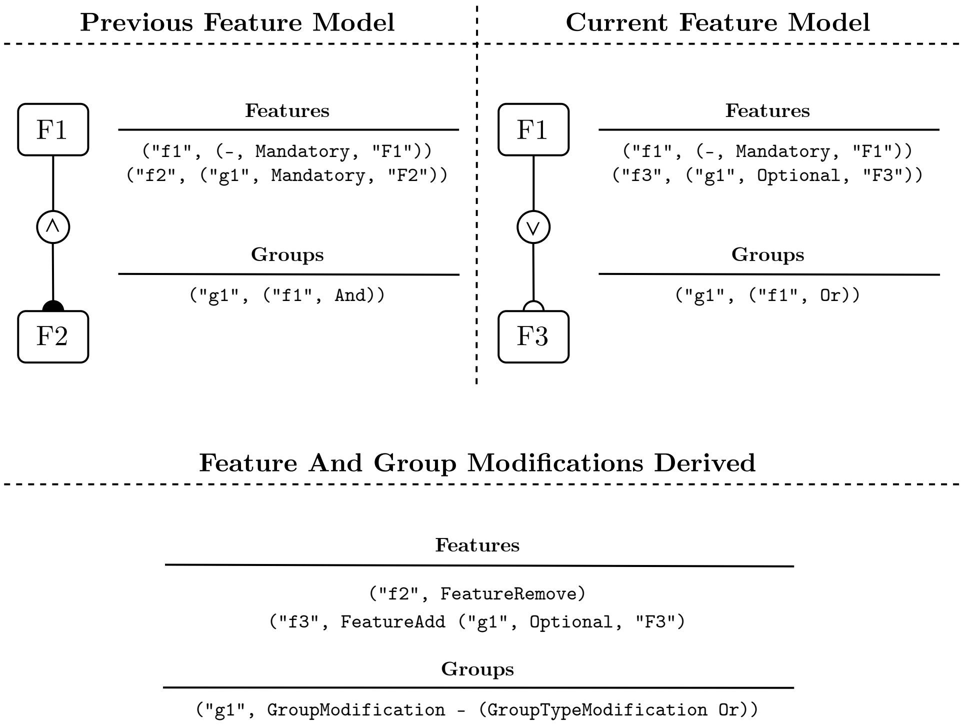


Figure 4.3: Result of executing `diffFeatureModels` on two feature models

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 will combine 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 don't 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 appeared in both feature models, we are using 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 of transforming a evolution plan into a merge-ready evolution plan using our two functions `flattenEvolutionPlan` and `deriveModifications`. However, since we want to merge two evolution plans into a single evolution plan, we have to figure out an approach to detecting, comparing and merging 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 18, 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 26. 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.

4.3.2 Representing Changes Between Versions

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

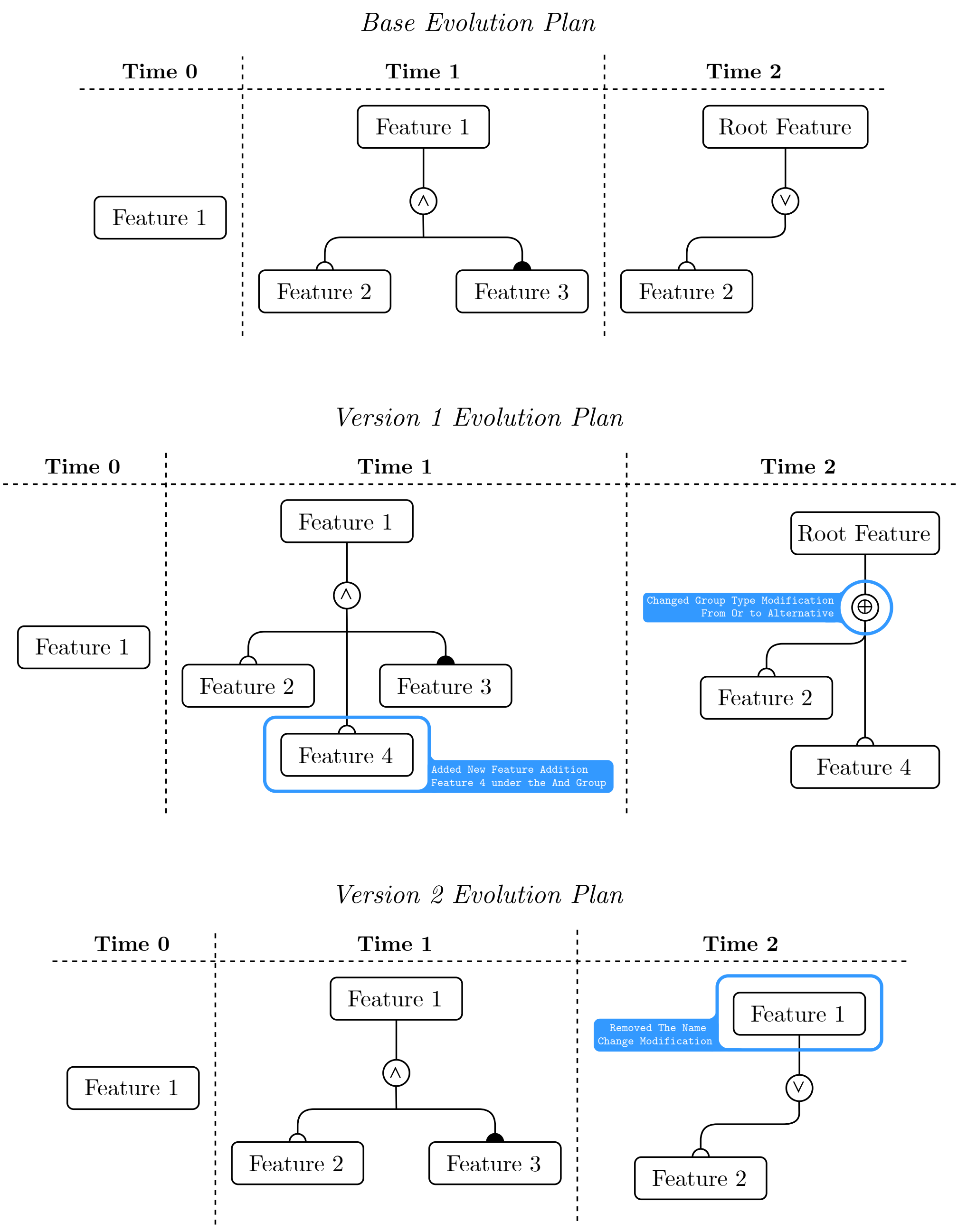


Figure 4.4: A simple example of three evolution plans, as input of the three-way merge algorithm

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 has 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 are removing a feature in 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 37

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

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

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

type FeatureDiffResult =
  SingleDiffResult FeatureModification

type GroupDiffResult =
  SingleDiffResult GroupModification

-- Every possible combination that a feature
-- or group change could be modified
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

data Version
  = V1
```

```haskell
  | 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 modification remained 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 three evolution plans into a 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 =
  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, and some times 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 `getModificationForTime`, which returns the modifications for a given time. Since the different plans doesn't necessarily include all the same time points, the function will create an empty list of modifications when a time point isn't present in the given evolution plan.

```
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 will handle the transformation combining modifications into the `DiffResult` type. Combining the modifications of groups follow the same general process as with feature modifications.

Combining the different `Maps` follow a similar approach as we did deriving the modifications between feature models (See Section 4.2.3 on page 40), 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 ->
  Modifications ->
  Modifications ->
```

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

```haskell
  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)
            )
    inBoth baseMod derivedResult =
      case derivedResult of
```

```haskell
        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 off 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`.

We can notice from the three-way merge algorithm outline from Figure 3.2 on page 18 that the `mergePlan` was split into two parts, `createMergePlan` and `unifyMergePlan`. This had several benefits. Both functions had clearly defined purposes. The only purpose of `createMergePlan` was 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 `MergeEvolutionPlan` also serves as documentation, letting the reader see more clearly what cases has 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 19), 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 in to 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 `MergeEvolutionPlan`. This contains all information about what modification was originally in the base evolution plan, as well as what change 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 will attempt to unify every time point, change and modification into our `FlatModificationEvolutionPlan`. If it produces a merge conflict somewhere, the conflict would automatically propagate 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, the conflict is propagated to the top level.

4.4.2 Specifying the Unification of the Merge Plan

The first three functions will 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 changed these modifications. The functions will go through each feature and group, and call the `unifySingleDiffResult` to unify the changes to a single feature or group.

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

```haskell
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 `modificationType`. This function is general and works for both features and groups, meaning we will get either a `FeatureModification` or a `GroupModification`. 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 we 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 changed a feature or group modification, we only want to raise an error if the changes were 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 47, 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:

```haskell
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 returned what we wanted. From both derived versions, there was 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 transform to a `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 would 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 will design an algorithm that makes sure that each modification is valid. In order to do so, we will convert our `FlatModificationEvolutionPlan` representation back to our normal form for evolution plans, namely `TreeUserEvolutionPlan`. Doing so will let us see the effects of applying each modification, and ensuring the resulting feature models and evolution plan is correct.

Looking back at the algorithm outline in Figure 3.2 on page 18, we can see that there are three remaining steps in the algorithm; `integrateModifications`, `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 algorithm 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 will not check constrains immediately after applying a modification. Instead, we will note the potential points where our result might be invalid, and check those after every modification has been integrated at a certain time.

When applying modifications, we generate a list of *dependencies* that we will 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 will check every dependency and either accept or reject the feature model.

Below, we will define the code necessary to intertwine both the integration and soundness checking of modifications. Since integrating and checking modifications both can lead to conflicts, the functions will return `Either Conflict value` instead of just `value`. Using do-notation, the monadic structure will make 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 is integrating 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 that handles conflicts, writing dependencies and returning the merged time point without to 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 returns both the next time point as well as the dependencies that needs 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 raise a conflict. Upon failure, the do-notation and monadic structure will 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 modifications for a single time point, the `integrateSinglePlan` function is called. As discussed, the modifications for a single time point has 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 will stop and the conflict will be 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 will be given a modification on a feature or group, as well as the current feature model. Based on the type of modification, the function will write dependencies with the `tell` function as well as incorporate the modification in the feature model. If a conflict occurs somewhere, the `throwError` function will be used to short circuit the function and return the conflict to the top level.

```
integrateFeature ::
  Time ->
  FeatureId ->
  FlatFeatureModel ->
  FeatureModification ->
  WriterT [Dependency] (Either Conflict) FlatFeatureModel
```

```haskell
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)
```

```haskell
where
  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
```

```haskell
            [Dependency]
            (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
```

```haskell
            time
            (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
```

```haskell
    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 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 don't 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 can not add then remove a feature in the same time point. The local conflicts can thus be reported immediately without checking the rest of the modifications at the time point.

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

4.5.3 Dependencies

With local conflicts, the conflicts were local to the single feature or group at hand. These conflicts could be raised without knowing what the rest of the modifications were. However, some of the modifications relied on knowing the state of other features or groups, which prevents us from reporting these conflicts immediately. Some modifications, i.e. adding a feature, relied on parent nodes to exist. However, to know if the parent existed or not, we would 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 marks 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 4.5.3

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 could encounter is the *global* conflict. For a given time point, we generated a list of dependencies we needed 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 `checkGlobalConflict`. The function will either succeed with the correct feature model, or fail with a list of dependencies that did 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
```

```haskell
                  ^?! 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 `FlatModificationEvolutionPlan` 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 47, we pick up the example from where we left off. With the `FlatModificationEvolutionPlan` representation of the merged plan, we use `integrateAndCheckModifications` to apply the modifications and check the result for soundness. This results in a list of feature models, in our `FlatUserEvolutionPlan` 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 is visualized below.

```
generatedDependencies :: [(Time, [Dependency])]
generatedDependencies =
```

```
[
  ( 0
  ,
    [ 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")
    ]
  )
,
  ( 1
  ,
    [ 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.

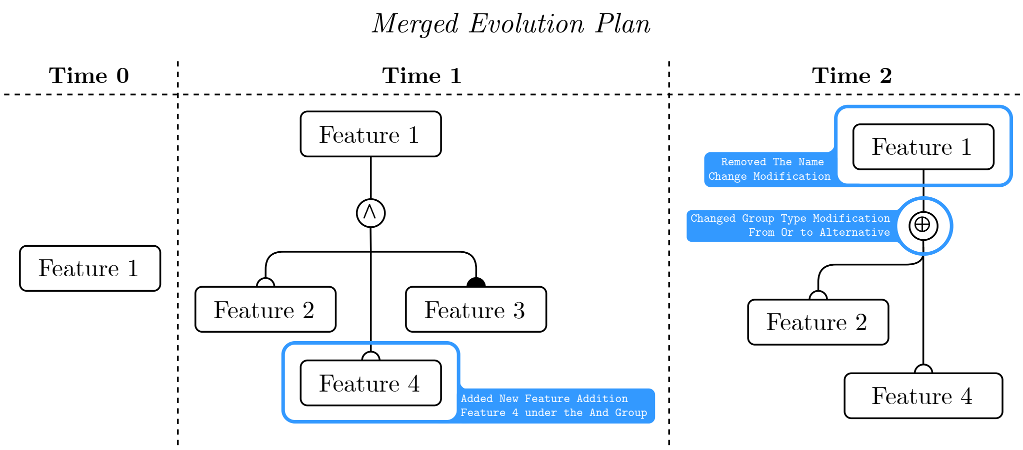


Figure 4.5: A visualization of the merged result of our simple example

Chapter 5

Command-Line Interface and Visualization Tools

[[TODO: bla bla about tools. cli and visualization tool.]]

5.1 Command-Line Interface

The evolution plan merger defines a command-line interface, `epmerge`, which will do 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

Usage: epmerge ( --generateOne EXAMPLENAME
               | --generateAll
               | --fromFile FILENAME
               )
```

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

```
               [-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 will 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.

The command-line interface has three basic 'Modes'. 'GenerateAll', 'GenerateOne' and 'FromFile'.

- `GenerateAll` will simply run all the examples in the code. This includes some sound examples and some erroneous examples. The erroneous examples consists 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. The merger will run the merger on the specified example. This mode can be

run using `epmerge --generateOne EXAMPLENAME`. 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 behaviour 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

[[TODO: elm, files generated from cli. display the eps as a list of trees, with each tree automaticcally drawn using a svg library. designed to

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

help explore the three-way merge algorithm, and its inputs and results, svg calculating and drawing each tree to do automatically, showcase a screendump of web page, explaining different elements. used to explore the different examples and edgecases of my tests, note about the visualization tool, how it differs from the paper. mandatory/optional are modeled as colors not dots above the feature. Results are generated and drawn automatically, what we display is simply a screenshot]]

Part III

Case Study and Conclusion

Chapter 6

Case Study – Vending Machine

In this chapter, we will apply the three-way merge tool on an example inspired from the real world. The example models the evolution of a vending machine software product line. The evolution plan models 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 will 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 will showcase two slightly different examples, an example that results in a sound, well-formed evolution plan, and one that results in a conflict. 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 is then serialized and written to a file. The command-line interface will also generate data which is passed to the visualization tool, which visualizes the input and output of the algorithm.

6.1 A Sound Example

[[TODO: sound example. show the base ep. summarize the changes in v1 and v2. changes can be harmonized, show the result]].

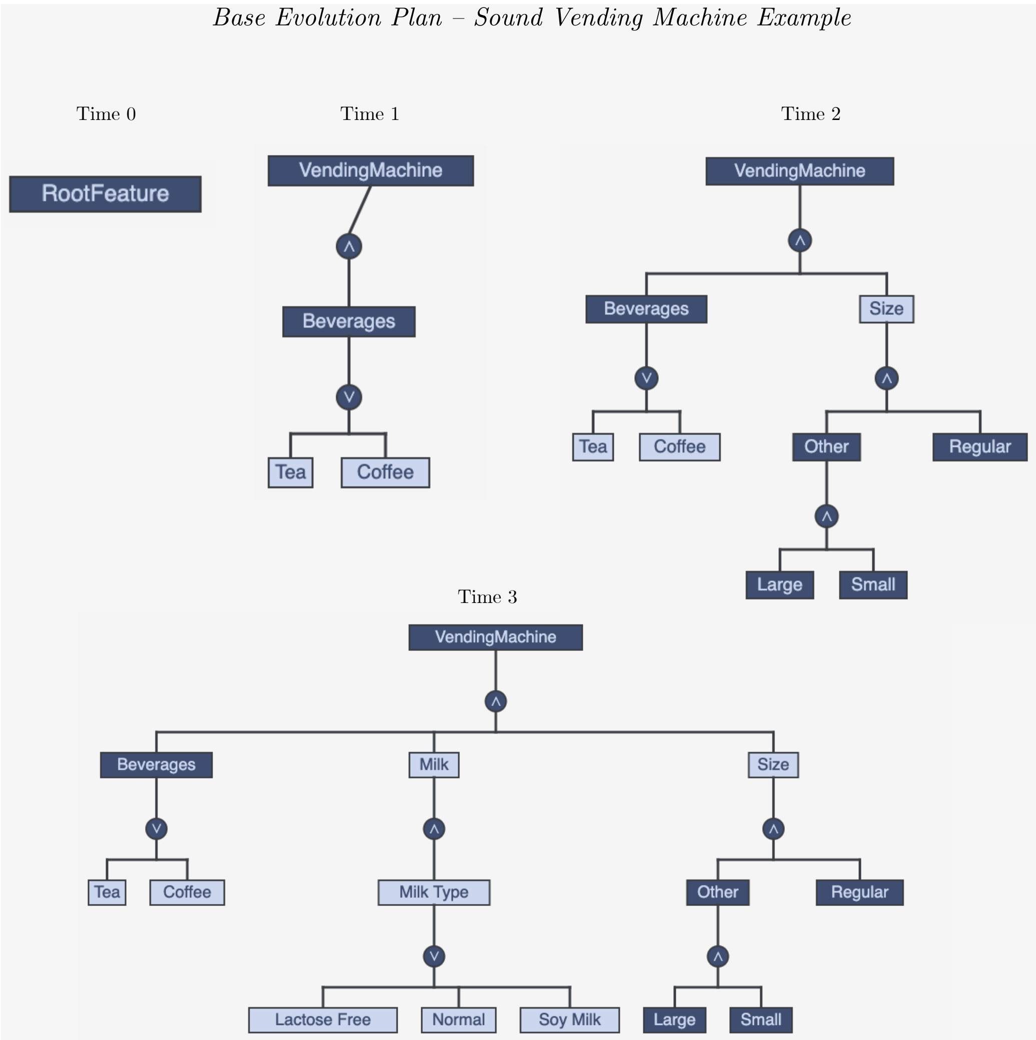


Figure 6.1: Sound Vending Machine Example - Base Evolution Plan

6.2 An Unsound Example

[[TODO: unsound example, note that all input eps are sound, this is an assumption. show base ep if not the same as before!. show the base ep. summarize the changes in v1 and v2. maybe say what is different from our sound example. changes cannot be harmonized, show the result]]

[[TODO: OlDTODOwrite about the vending machine example. Write about how the entire example is done. from cli parsing, converting to right representation, merging, checking, converting back, writing to file. Then how the frontend visualization tool parses the result and displays the tree as an interactive thing. sound vs unsound examples]]

Chapter 7

Conclusion and Future Work

[[TODO: konklusjon bør si hvordan det jeg har gjort addresserer forskningsspørsmålene forskningsspørsmålene kan være større en bidraget how to ensure sound plan and merge of sound plans future work kan være flere ting å se på for å belyse forskningsspm]]

Bibliography

[1] Don S. Batory. "Feature Models, Grammars, and Propositional Formulas". In: *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*. Ed. by J. Henk Obbink and Klaus Pohl. Vol. 3714. Lecture Notes in Computer Science. Springer, 2005, pp. 7–20. DOI: 10.1007/11554844_3. URL: https://doi.org/10.1007/11554844%5C_3.

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

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

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

[5] Linda M. Northrop. "SEI's Software Product Line Tenets". In: *IEEE Softw.* 19.4 (2002), pp. 32–40. DOI: 10.1109/MS.2002.1020285. URL: https://doi.org/10.1109/MS.2002.1020285.

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

[7] Eirik Halvard Sæther et al. "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*. Ed. by Roberto Erick Lopez-Herrejon. ACM, 2020, 8:1–8:12. DOI: 10.1145/3382025.3414964. URL: https://doi.org/10.1145/3382025.3414964.

[8] Walter F. Tichy. "Tools for Software Configuration Management". In: *Proceedings of the International Workshop on Software Version and*

Configuration Control, January 27-29, 1988, Grassau, Germany. Ed. by Jürgen F. H. Winkler. Vol. 30. Berichte des German Chapter of the ACM. Teubner, 1988, pp. 1–20.