

# Three Way Merge for Feature Model Evolution Plans

Eirik Halvard Sæther



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

Department of Informatics  
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021



# **Three Way Merge for Feature Model Evolution Plans**

Eirik Halvard Sæther

© 2021 Eirik Halvard Sæther

Three Way Merge for Feature Model Evolution Plans

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

Printed: Reprosentralen, University of Oslo

# Acknowledgements

Thanks to Ida

Thanks to Ingrid and Crystal

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

Thanks to ifi, all the people i have met

Thanks to friends and family



# Abstract

[[TODO: write abstract]]

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

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

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

# Contents



# List of Figures

# List of Tables

# Preface

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

# Chapter 1

## Introduction

### 1.1 Motivation

### 1.2 Problem Statement

### 1.3 Research Questions

### 1.4 Contributions

what i have done

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

[[TODO: WRITE]]

### 1.5 Chapter Overview

Chapter ?? something about background

Chapter ?? something about the bigboy algo

[[TODO: WRITE]]

## 1.6 Project Source Code

All the source code from the master thesis can be found on Github<sup>1</sup>.

---

<sup>1</sup><https://github.com/eirikhalvard/master-thesis>

# Chapter 2

## Background

### 2.1 Software product lines

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

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

### 2.2 Feature Models

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

A visual representation of a feature model can be seen in Figure ??.

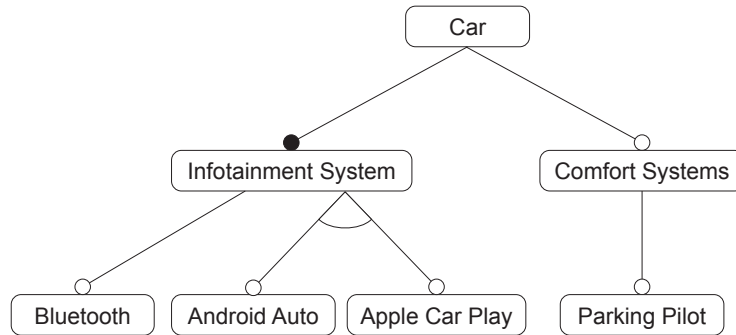


Figure 2.1: Example feature model

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

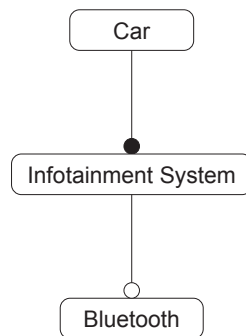
## 2.3 Evolution planning

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

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

*Evolution plans* lets us model a sequence of feature models, which represents the current and all planned future versions of the feature model. Each feature model represents the product line in a point in

time, which could have varying validity, from a week from now to a year. Since the next feature model is derived from the previous one, we can represent the evolution plan as an initial feature model, as well as a sequence of *points*, where each point is a set of operations to perform on the previous feature model to achieve the current one. The operations vary from changing, adding or deleting features or groups from the feature model.



#### At time 1:

---

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

#### At time 2:

---

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

Figure 2.2: An example evolution plan

An example of an evolution plan can be seen in Figure ???. 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 ??

## 2.4 Version Control Systems

*Software configuration mechanisms* is the discipline of managing the evolution of large and complex software systems [cite:software configuration management]. *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 [cite:tom`mens`software`merging`survey] categorizes and describes different aspect of version control systems and software merging techniques. Two-way and three-way merging differentiates between how many versions of the artifact you are comparing. Different representations of the merge artifact can be categorized in textual, syntactic, semantic or structural merging. State-based merge techniques uses delta algorithms to compute differences between revisions while change-based techniques keeps track of the exact operations that were performed between the revisions.

### 2.4.1 Two-way vs three-way merging

When merging different versions of a piece of software, we differentiate between *two-way* and *three-way* merging. Two-way merging merges the two versions without taking a common ancestor into account. Three-way merging on the other hand, uses a common ancestor as a reference point, to know how the different versions were changed. The latter technique is more powerful and produces more accurate merges, because the merge will know extra information from the common ancestor.

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

### 2.4.2 Textual merging

Textual merging views the software artifacts as unstructured text files. There exist several granularities of what is considered one unit, but *line-*

*based merging* is probably the most common textual merge. Line-based merging techniques computes the difference between files by comparing equality over the lines. This has several implications, like adding a single space after a line is considered a deletion of the old line and addition of the new. This coarse granularity often leads to unnecessary and confusing conflicts. Changing the indentation or other formatting differences often lead to unnecessary conflicts.

To exemplify this, consider the two versions of a Python program, Listing ?? and Listing ?. 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 [cite:fast'algo'for'lcs], we are able to calculate the difference between the two files by calculating the longest common subsequence. As seen in the result (Listing ?), 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

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

Listing 2: Code diff 2

As discussed, text-based merge techniques often provide inferior results, however, they have several advantages in terms of efficiency and generality. The algorithm is general enough to work well for different programming languages, documentation, markup files, configuration files, etc.

```

<  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

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

### 2.4.3 Syntactic Merging

*Syntactic merging* [cite:syntactic'software'merging] 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 ?? and Listing ?? as a parse tree or abstract syntax tree, we can avoid merge conflicts.

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

### 2.4.4 Semantic Merging

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

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

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

## 2.5 Haskell and Algebraic Data Types

## **Chapter 3**

# **Formal Semantics of Feature Model Evolution Plans**

# Chapter 4

## Three Way Merge Algorithm

### 4.1 Algorithm Overview

#### 4.1.1 Three-Way Merging of Evolution Plans

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

#### 4.1.2 Soundness Assumption

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

#### 4.1.3 Algorithm Phases

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

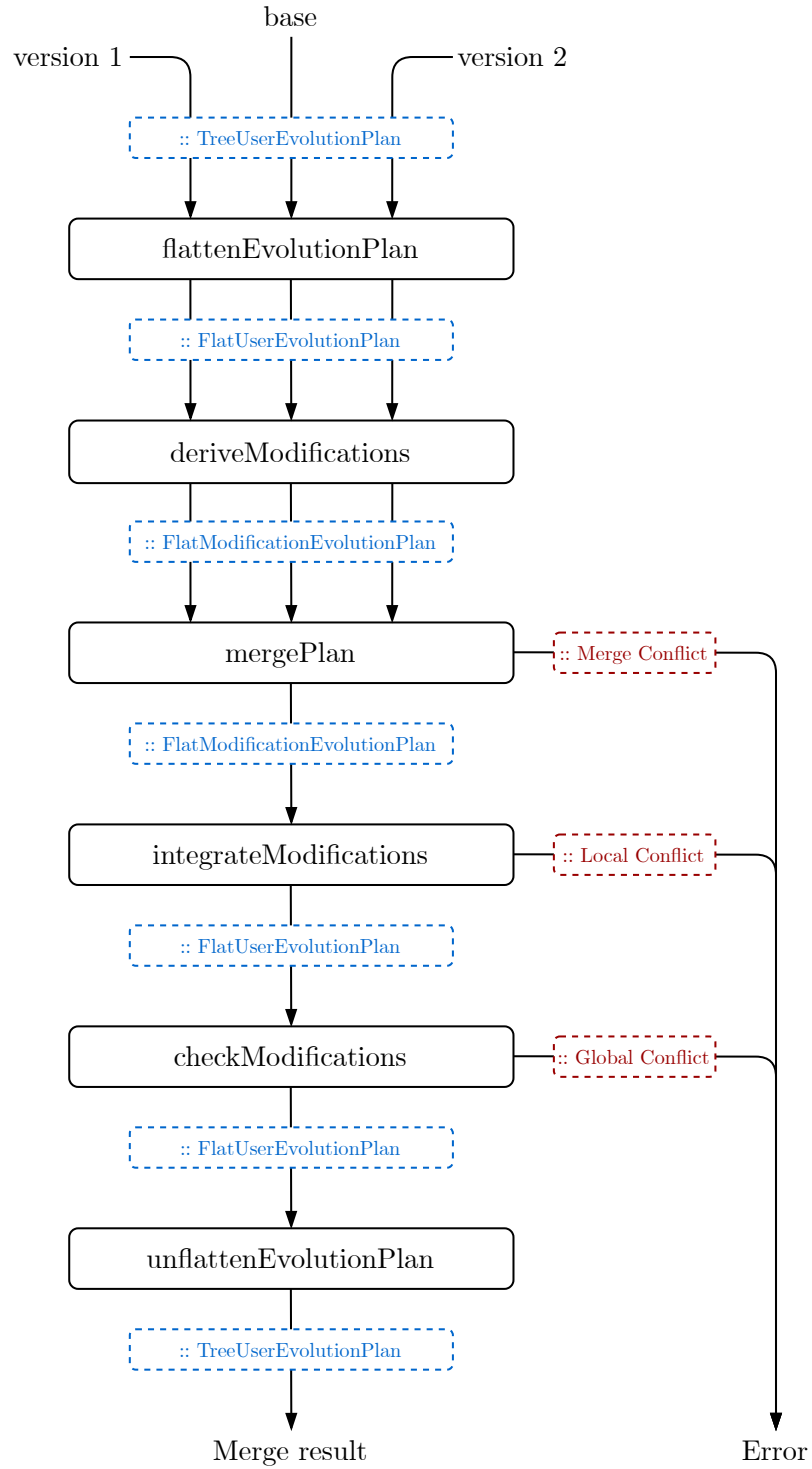


Figure 4.1: Outline of the three-way merge algorithm

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

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 ??

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

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 ??

#### 4.1.4 Conflicts

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

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



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

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

## **4.2 Converting To a Suitable Representation**

### **4.2.1 Representing Feature Models**

### **4.2.2 Representing Evolution Plans**

## **4.3 Detecting the Changes Between Versions**

## **4.4 Merging Intended Changes**

## **4.5 Ensuring structural and semantic soundness of the merge result**

## **Chapter 5**

### **Conclusion and Future Work**

# Appendices

# Appendix A

## Types

```
1 {-# LANGUAGE DataKinds #-}
2 {-# LANGUAGE DeriveFunctor #-}
3 {-# LANGUAGE DeriveGeneric #-}
4 {-# LANGUAGE DerivingVia #-}
5 {-# LANGUAGE DuplicateRecordFields #-}
6 {-# LANGUAGE FlexibleContexts #-}
7 {-# LANGUAGE FlexibleInstances #-}
8
9 module Types where
10
11 import qualified Data.Map as M
12 import qualified Data.Set as S
13
14 import Data.Aeson
15 import Deriving.Aeson
16 import Deriving.Aeson.Stock
17 import GHC.Generics
18
19 -----
20 --                                     Feature Models
21 --                                     ↔
22 -----
23 type FeatureId = String
24
25 type GroupId = String
26
27 --- Tree Structured Feature Model ---
28
29 data TreeFeatureModel = TreeFeatureModel
```

```

30     { _rootFeature :: TreeFeature
31     }
32     deriving (Show, Eq, Read, Generic)
33     deriving
34         (FromJSON, ToJSON)
35     via Prefixed "_" TreeFeatureModel
36
37 data TreeFeature = TreeFeature
38     { _id :: FeatureId
39     , _featureType :: FeatureType
40     , _name :: String
41     , _groups :: S.Set TreeGroup
42     }
43     deriving (Show, Eq, Read, Ord, Generic)
44     deriving
45         (FromJSON, ToJSON)
46     via Prefixed "_" TreeFeature
47
48 data TreeGroup = TreeGroup
49     { _id :: GroupId
50     , _groupType :: GroupType
51     , _features :: S.Set TreeFeature
52     }
53     deriving (Show, Eq, Read, Ord, Generic)
54     deriving
55         (FromJSON, ToJSON)
56     via Prefixed "_" TreeGroup
57
58 --- Flat Structured Feature Model ---
59
60 data FlatFeatureModel = FlatFeatureModel
61     { _rootId :: FeatureId
62     , _features :: M.Map FeatureId FlatFeature
63     , _groups :: M.Map GroupId FlatGroup
64     }
65     deriving (Show, Eq, Read, Generic)
66     deriving
67         (FromJSON, ToJSON)
68     via Prefixed "_" FlatFeatureModel
69
70 data FlatFeature = FlatFeature
71     { _parentGroupId :: Maybe GroupId
72     , _featureType :: FeatureType
73     , _name :: String
74     }

```

```

75     deriving (Show, Eq, Read, Generic)
76     deriving
77         (FromJSON, ToJSON)
78         via Prefixed "_" FlatFeature
79
80 data FlatGroup = FlatGroup
81     { _parentFeatureId :: FeatureId
82     , _groupType :: GroupType
83     }
84     deriving (Show, Eq, Read, Generic)
85     deriving
86         (FromJSON, ToJSON)
87         via Prefixed "_" FlatGroup
88
89 data FeatureType
90     = Optional
91     | Mandatory
92     deriving (Show, Eq, Read, Ord, Generic)
93     deriving
94         (FromJSON, ToJSON)
95         via Prefixed "_" FeatureType
96
97 data GroupType
98     = And
99     | Or
100    | Alternative
101    deriving (Show, Eq, Read, Ord, Generic)
102    deriving
103        (FromJSON, ToJSON)
104        via Prefixed "_" GroupType
105
106 -----
107 --                                     Evolution Plans
108 --                                     -----
109
110 -- Four different types of evolution plan representations.
111 -- We categorize them in
112 -- two categories. User evolution plans and Transformation
113 -- evolution plans
114 --
115 --     User Evolution Plans:
116 --     Represents the evolution plan as a list of feature
117 --     models, where each

```

```

115 --      feature model is coupled with a time point. In this
116 --      ↪ representation the
117 --      exact changes between each feature model is implicit
118 --      ↪ as the difference
119 --      between each pair of feature models
120 --      Transformation Evolution Plans:
121 --      Represents the evolution plan as an initial model,
122 --      ↪ together with a list
123 --      of plans, where each plan is a time point and a
124 --      ↪ transformation. The
125 --      transformation describes how the previous feature
126 --      ↪ model should be
127 --      transformed in order to achieve the feature model at
128 --      ↪ the given time
129 --      point. We define two different types of
130 --      ↪ transformations, namely
131 --      Modification level and merge level modifications.
132 --
133 --      Modification Transformation:
134 --      Represents the transformation as a set of
135 --      ↪ modifications. This
136 --      representation guarantees that each there are no
137 --      ↪ conflicting
138 --      modifications, i.e. moving a feature twice. This
139 --      ↪ allows for merging
140 --      the modifications in an arbitrary ordering, since
141 --      ↪ no modifications
142 --      shadow others, etc.
143 --      Merge Transformation:
144 --      The merge level transformation represents the
145 --      ↪ "planned"
146 --      transformations from both versions in the merge.
147 --      ↪ The transformation
148 --      is essentially the union of the modifications of
149 --      ↪ version 1 and
150 --      version 2. In this representation, a feature might
151 --      ↪ be planned to be
152 --      changed, added or removed in several versions,
153 --      ↪ which this
154 --      representation encodes.
155
156 type TreeUserEvolutionPlan = UserEvolutionPlan TreeFeatureModel
157
158 type FlatUserEvolutionPlan = UserEvolutionPlan FlatFeatureModel

```

```

144 type FlatModificationEvolutionPlan = ModificationEvolutionPlan
    ↳ FlatFeatureModel
145
146 type Time = Int
147
148 data UserEvolutionPlan featureModel = UserEvolutionPlan
149   { _timePoints :: [TimePoint featureModel]
150   }
151   deriving (Show, Eq, Read, Generic)
152   deriving
153     (FromJSON, ToJSON)
154     via Prefixed "_" (UserEvolutionPlan featureModel)
155
156 data TimePoint featureModel = TimePoint
157   { _time :: Time
158   , _featureModel :: featureModel
159   }
160   deriving (Show, Eq, Read, Generic)
161   deriving
162     (FromJSON, ToJSON)
163     via Prefixed "_" (TimePoint featureModel)
164
165 data TransformationEvolutionPlan transformation featureModel =
166   ↳ TransformationEvolutionPlan
167   { _initialTime :: Time
168   , _initialFM :: featureModel
169   , _plans :: [Plan transformation]
170   }
171   deriving (Show, Eq, Read, Generic)
172   deriving
173     (FromJSON, ToJSON)
174     via Prefixed "_" (TransformationEvolutionPlan
175       ↳ transformation featureModel)
176
177 data Plan transformation = Plan
178   { _timePoint :: Time
179   , _transformation :: transformation
180   }
181   deriving (Show, Eq, Read, Generic)
182   deriving
183     (FromJSON, ToJSON)
184     via Prefixed "_" (Plan transformation)
185
186 type ModificationEvolutionPlan featureModel =
187   ↳ TransformationEvolutionPlan Modifications featureModel

```



```

185
186 type MergeEvolutionPlan featureModel =
187     ↳ TransformationEvolutionPlan DiffResult featureModel
188
189 -----
190
191
192 Transformation Types
193
194 -----
195
196 --- MODIFICATIONS ---
197
198 -- Modifications vs Changes
199 -- We have two levels of changes. To differentiate between
200 ↳ the two, we will use
201 -- the name Modification or Change in order to separate the
202 ↳ two
203
204 --
205 -- Modifications:
206 -- Modifications are the actual changes between two feature
207 ↳ models. For
208 -- example, If a feature was removed or added, we will call
209 ↳ this "change" as
210 -- a Modification
211 --
212 -- Changes:
213 -- Changes are relevant to the diff-algorithm and its
214 ↳ output, and refer to
215 -- the meta-level changes on modifications. If a base
216 ↳ version included
217 -- a Modification, i.e. an addition of a feature, one of
218 ↳ the derived versions
219 -- could remove this modification The derived version has
220 ↳ then Changed
221 -- a modification. So Change-names is reserved for these
222 ↳ meta-level changes
223
224 --- Modifications between featuremodels ---
225
226 data Modifications = Modifications
227     { _features :: M.Map FeatureId FeatureModification
228     , _groups  :: M.Map GroupId GroupModification
229     }
230
231 deriving (Show, Eq, Read, Generic)
232 deriving
233     (FromJSON, ToJSON)

```

```

219     via Prefixed "_" Modifications
220
221 data FeatureModification
222   = FeatureAdd GroupId FeatureType String
223   | FeatureRemove
224   | FeatureModification
225     (Maybe FeatureParentModification)
226     (Maybe FeatureTypeModification)
227     (Maybe FeatureNameModification)
228   deriving (Show, Eq, Read, Generic)
229   deriving
230     (FromJSON, ToJSON)
231   via Prefixed "_" FeatureModification
232
233 data FeatureParentModification
234   = FeatureParentModification GroupId
235   deriving (Show, Eq, Read, Generic)
236   deriving
237     (FromJSON, ToJSON)
238   via Prefixed "_" FeatureParentModification
239
240 data FeatureNameModification
241   = FeatureNameModification String
242   deriving (Show, Eq, Read, Generic)
243   deriving
244     (FromJSON, ToJSON)
245   via Prefixed "_" FeatureNameModification
246
247 data FeatureTypeModification
248   = FeatureTypeModification FeatureType
249   deriving (Show, Eq, Read, Generic)
250   deriving
251     (FromJSON, ToJSON)
252   via Prefixed "_" FeatureTypeModification
253
254 data GroupModification
255   = GroupAdd FeatureId GroupType
256   | GroupRemove
257   | GroupModification
258     (Maybe GroupParentModification)
259     (Maybe GroupTypeModification)
260   deriving (Show, Eq, Read, Generic)
261   deriving
262     (FromJSON, ToJSON)
263   via Prefixed "_" GroupModification

```

```

264
265 data GroupParentModification
266     = GroupParentModification FeatureId
267     deriving (Show, Eq, Read, Generic)
268     deriving
269         (FromJSON, ToJSON)
270     via Prefixed "_" GroupParentModification
271
272 data GroupTypeModification
273     = GroupTypeModification GroupType
274     deriving (Show, Eq, Read, Generic)
275     deriving
276         (FromJSON, ToJSON)
277     via Prefixed "_" GroupTypeModification
278
279 --- DIFF RESULT ---
280
281 -- The diff result from the all the changes in the entire
282   ↪ time point for all
283 -- versions of the model
284 data DiffResult = DiffResult
285     { _features :: M.Map FeatureId FeatureDiffResult
286     , _groups   :: M.Map GroupId GroupDiffResult
287     }
288     deriving (Show, Eq, Read, Generic)
289     deriving
290         (FromJSON, ToJSON)
291     via Prefixed "_" DiffResult
292
293 type FeatureDiffResult =
294     SingleDiffResult FeatureModification
295
296 type GroupDiffResult =
297     SingleDiffResult GroupModification
298
299 -- Every possible combination that a feature- or group change
300   ↪ could be modified
301 data SingleDiffResult modificationType
302     = NoChange modificationType
303     | ChangedInOne Version (OneChange modificationType)
304     | ChangedInBoth (BothChange modificationType)
305     deriving (Show, Eq, Read, Generic)
306     deriving
307         (FromJSON, ToJSON)
308     via Prefixed "_" (SingleDiffResult modificationType)

```

```

307
308 data OneChange modificationType
309   = OneChangeWithBase
310     modificationType -- Base modification
311     (RemovedOrChangedModification modificationType) --
312       ↳ Derived (V1 or V2) modification
313   | OneChangeWithoutBase
314     (AddedModification modificationType) -- Derived (V1 or
315       ↳ V2) modification
316 deriving (Show, Eq, Read, Generic)
317 deriving
318   (FromJSON, ToJSON)
319   via Prefixed "_" (OneChange modificationType)
320
321 data BothChange modificationType
322   = BothChangeWithBase
323     modificationType -- Base modification
324     (RemovedOrChangedModification modificationType) -- V1
325       ↳ modification
326     (RemovedOrChangedModification modificationType) -- V2
327       ↳ modification
328   | BothChangeWithoutBase
329     (AddedModification modificationType) -- V1 modification
330     (AddedModification modificationType) -- V2 modification
331 deriving (Show, Eq, Read, Generic)
332 deriving
333   (FromJSON, ToJSON)
334   via Prefixed "_" (BothChange modificationType)
335
336 data RemovedOrChangedModification modificationType
337   = RemovedModification
338   | ChangedModification modificationType
339 deriving (Show, Eq, Read, Generic)
340 deriving
341   (FromJSON, ToJSON)
342   via Prefixed "_" (RemovedOrChangedModification
343     ↳ modificationType)
344
345 data AddedModification modificationType
346   = AddedModification modificationType
347 deriving (Show, Eq, Read, Generic)
348 deriving
349   (FromJSON, ToJSON)
350   via Prefixed "_" (AddedModification modificationType)

```

```

347 data Version
348   = V1
349   | V2
350   deriving (Show, Eq, Read, Generic)
351   deriving
352     (FromJSON, ToJSON)
353     via Prefixed "_" Version
354
355 -----
356 --                                     Merge Input / Output
357 --                                     ↔
358 -----
359 data MergeInput
360   = TreeUser (MergeInputData TreeUserEvolutionPlan)
361   | FlatUser (MergeInputData FlatUserEvolutionPlan)
362   | FlatModification (MergeInputData
363     ↔ FlatModificationEvolutionPlan)
364   deriving (Show, Eq, Read)
365
366 data MergeInputData evolutionPlan = MergeInputData
367   { _name :: String
368   , _base :: evolutionPlan
369   , _v1 :: evolutionPlan
370   , _v2 :: evolutionPlan
371   , _maybeExpected :: Maybe (MergeResult evolutionPlan)
372   }
373   deriving (Show, Eq, Read, Generic, Functor)
374   deriving
375     (FromJSON, ToJSON)
376     via Prefixed "_" (MergeInputData evolutionPlan)
377
378 type MergeOutput = Either Conflict
379   ↔ (FlatModificationEvolutionPlan, FlatUserEvolutionPlan)
380
381 type MergeResult evolutionPlan = Either Conflict evolutionPlan
382
383 -----
384 --                                     Elm Data Serialization
385 --                                     ↔
386 -----
387 data ElmDataExamples = ElmDataExamples
388   { _examples :: [ElmMergeExample]
389   }

```

```

388     deriving (Show, Eq, Read, Generic)
389     deriving
390         (FromJSON, ToJSON)
391         via Prefixed "_" ElmDataExamples
392
393 data ElmMergeExample = ElmMergeExample
394     { _name :: String
395     , _evolutionPlans :: [ElmNamedEvolutionPlan]
396     }
397     deriving (Show, Eq, Read, Generic)
398     deriving
399         (FromJSON, ToJSON)
400         via Prefixed "_" ElmMergeExample
401
402 data ElmNamedEvolutionPlan = ElmNamedEvolutionPlan
403     { _name :: String
404     , _mergeData :: Either String TreeUserEvolutionPlan
405     }
406     deriving (Show, Eq, Read, Generic)
407     deriving
408         (FromJSON, ToJSON)
409         via Prefixed "_" ElmNamedEvolutionPlan
410
411 -----
412 --                                     Conflict
413 --                                     ↔
414 -----
415 data Conflict
416     = Merge Time MergeConflict
417     | Local Time LocalConflict
418     | Global Time GlobalConflict
419     | Panic Time String
420     deriving (Show, Eq, Read, Generic)
421     deriving
422         (FromJSON, ToJSON)
423         via Prefixed "_" Conflict
424
425 data MergeConflict
426     = FeatureConflict FeatureId (BothChange FeatureModification)
427     | GroupConflict GroupId (BothChange GroupModification)
428     deriving (Show, Eq, Read, Generic)
429     deriving
430         (FromJSON, ToJSON)
431         via Prefixed "_" MergeConflict

```

```

432
433 data LocalConflict
434     = FeatureAlreadyExists FeatureModification FeatureId
435     | FeatureNotExists FeatureModification FeatureId
436     | GroupAlreadyExists GroupModification GroupId
437     | GroupNotExists GroupModification GroupId
438 deriving (Show, Eq, Read, Generic)
439 deriving
440     (FromJSON, ToJSON)
441     via Prefixed "_" LocalConflict
442
443 data GlobalConflict
444     = FailedDependencies [Dependency]
445 deriving (Show, Eq, Read, Generic)
446 deriving
447     (FromJSON, ToJSON)
448     via Prefixed "_" GlobalConflict
449
450 data Dependency
451     = FeatureDependency FeatureModification FeatureDependencyType
452     | GroupDependency GroupModification GroupDependencyType
453 deriving (Show, Eq, Read, Generic)
454 deriving
455     (FromJSON, ToJSON)
456     via Prefixed "_" Dependency
457
458 data FeatureDependencyType
459     = NoChildGroups FeatureId
460     | ParentGroupExists GroupId
461     | NoCycleFromFeature FeatureId
462     | FeatureIsWellFormed FeatureId
463     | UniqueName String
464 deriving (Show, Eq, Read, Generic)
465 deriving
466     (FromJSON, ToJSON)
467     via Prefixed "_" FeatureDependencyType
468
469 data GroupDependencyType
470     = NoChildFeatures GroupId
471     | ParentFeatureExists FeatureId
472     | NoCycleFromGroup GroupId
473     | GroupIsWellFormed GroupId
474 deriving (Show, Eq, Read, Generic)
475 deriving
476     (FromJSON, ToJSON)

```

```

477     via Prefixed "_" GroupDependencyType
478
479 -----
480 --                                     CLI OPTIONS
481 --                                     ↵
482 -----
483 data EvolutionPlanType
484     = TreeUserType
485     | FlatUserType
486     | FlatModificationType
487     deriving (Show, Eq, Read)
488
489 data Mode
490     = GenerateOne String
491     | GenerateAll
492     | FromFile FilePath
493     deriving (Show, Eq, Read)
494
495 data CliOptions = CliOptions
496     { _mode :: Mode
497     , _fromType :: EvolutionPlanType
498     , _toType :: EvolutionPlanType
499     , _print :: Bool
500     , _generateElm :: Bool
501     , _toFile :: Maybe FilePath
502     }
503     deriving (Show, Eq, Read)

```



# Appendix B

## Three Way Merge Algorithm

```
1  {-# LANGUAGE FlexibleContexts #-}
2
3  module ThreeWayMerge where
4
5  import Convertable
6  import Merge.CheckPlan (integrateAndCheckModifications)
7  import Merge.PlanMerging (createMergePlan, unifyMergePlan)
8  import Types
9
10 threeWayMerge ::
11   ConvertableInput inputEvolutionPlan =>
12   MergeInputData inputEvolutionPlan ->
13   MergeOutput
14 threeWayMerge (MergeInputData _ base v1 v2 _) = do
15   let mergePlan =
16       createMergePlan
17         (toFlatModification base)
18         (toFlatModification v1)
19         (toFlatModification v2)
20   mergedModificationPlan <- unifyMergePlan mergePlan
21   checkedUserFlatPlan <- integrateAndCheckModifications
22     ↪ mergedModificationPlan
23   return (mergedModificationPlan, checkedUserFlatPlan)
```

### B.1 Change Detection

```
1  module Merge.ChangeDetection where
2
```

```

3 import qualified Lenses as L
4 import Types
5
6 import Control.Lens
7 import qualified Data.Map as M
8 import qualified Data.Map.Merge.Lazy as Merge
9
10 -----
11 --                               Flatten Sound Evolution Plan
12 --                               ↪ --
13 -----
14 flattenSoundEvolutionPlan :: TreeUserEvolutionPlan ->
15   ↪ FlatUserEvolutionPlan
16 flattenSoundEvolutionPlan =
17   L.timePoints
18     . traversed
19     . L.featureModel
20     %~ flattenSoundFeatureModel
21
22 flattenSoundFeatureModel :: TreeFeatureModel ->
23   ↪ FlatFeatureModel
24 flattenSoundFeatureModel fm =
25   FlatFeatureModel
26     (fm ^. L.rootFeature . L.id)
27     (M.fromList features)
28     (M.fromList groups)
29   where
30     (features, groups) = flattenFeature Nothing (fm ^.
31       ↪ L.rootFeature)
32     flattenFeature mParentGroup (TreeFeature id featureType
33       ↪ name groups) =
34       ([ (id, FlatFeature mParentGroup featureType name) ], [])
35       <> foldMap (flattenGroup id) groups
36     flattenGroup parentFeature (TreeGroup id groupType
37       ↪ features) =
38       ([], [(id, FlatGroup parentFeature groupType)])
39       <> foldMap (flattenFeature (Just id)) features
40
41 -----
42 --                               Derive Sound Modifications
43 --                               ↪ --
44 -----

```

```

40 deriveSoundModifications :: FlatUserEvolutionPlan ->
    ↳ FlatModificationEvolutionPlan
41 deriveSoundModifications (UserEvolutionPlan timePoints) = case
    ↳ timePoints of
42   [] -> error "evolution plan has to have at least one time
    ↳ point!"
43   ((TimePoint initialTime initialFM) : restTimePoints) ->
44     TransformationEvolutionPlan
45       initialTime
46       initialFM
47       (zipWith timePointsToPlan timePoints restTimePoints)
48
49 timePointsToPlan ::
50   TimePoint FlatFeatureModel -> TimePoint FlatFeatureModel ->
    ↳ Plan Modifications
51 timePointsToPlan (TimePoint _ prevFM) (TimePoint currTime
    ↳ currFM) =
52   Plan currTime $ diffFeatureModels prevFM currFM
53
54 -- diffFeatureModels will derive every modification
55 diffFeatureModels :: FlatFeatureModel -> FlatFeatureModel ->
    ↳ Modifications
56 diffFeatureModels prevFM currFM =
57   Modifications
58     featureModifications
59     groupModifications
60 where
61   featureModifications =
62     Merge.merge
63       (Merge.mapMissing (\_ _ -> FeatureRemove))
64       ( Merge.mapMissing
65         ( \_ (FlatFeature mParent featureType name) ->
66           case mParent of
67             Nothing ->
68               error $
69                 "ERROR: When diffing two feature models,
        ↳ "
70                 ++ "the root feature is assumed to be
        ↳ the same in both version. "
71                 ++ "Since the root feature cannot be
        ↳ removed, there should never "
72                 ++ "be the case that the root feature
        ↳ was added"
73       Just parent -> FeatureAdd parent featureType
        ↳ name

```

```

74         )
75     )
76     ( Merge.zipWithMaybeMatched
77         ( \_
78             prev@(FlatFeature prevParent prevFeatureType
79                 ↪ prevName)
80             new@(FlatFeature newParent newFeatureType
81                 ↪ newName) ->
82                 if prev == new
83                 then Nothing
84                 else
85                     Just $
86                         FeatureModification
87                             ( case (prevParent, newParent) of
88                                 (Just prev, Just new) | prev /=
89                                     ↪ new -> Just
90                                     ↪ (FeatureParentModification
91                                         ↪ new)
92                                 -- NOTE: since the root is
93                                 ↪ assumed to never change,
94                                 -- we only record changes of
95                                 ↪ non-root features
96                                 _ -> Nothing
97                             )
98                             ( if prevFeatureType ==
99                                 ↪ newFeatureType
100                             then Nothing
101                             else Just
102                                 ↪ (FeatureTypeModification
103                                 ↪ newFeatureType)
104                             )
105                             ( if prevName == newName
106                             then Nothing
107                             else Just
108                                 ↪ (FeatureNameModification
109                                 ↪ newName)
110                             )
111                         )
112         )
113     )
114     (prevFM ^. L.features)
115     (currFM ^. L.features)
116 groupModifications =
117     Merge.merge
118         (Merge.mapMissing (\_ _ -> GroupRemove))
119         ( Merge.mapMissing

```

```

107         ( \_ (FlatGroup parent groupType) ->
108             GroupAdd parent groupType
109         )
110     )
111     ( Merge.zipWithMaybeMatched
112         ( \_
113             prev@(FlatGroup prevParent prevGroupType)
114             new@(FlatGroup newParent newGroupType) ->
115                 if prev == new
116                 then Nothing
117                 else
118                     Just $
119                         GroupModification
120                             ( if prevParent == newParent
121                               then Nothing
122                               else Just
123                                 ↪ (GroupParentModification
124                                   ↪ newParent)
125                             )
126                             ( if prevGroupType == newGroupType
127                               then Nothing
128                               else Just (GroupTypeModification
129                                   ↪ newGroupType)
130                             )
131     )
132     )
133     (prevFM ^. L.groups)
134     (currFM ^. L.groups)

```

## B.2 Plan Merging

```

1  module Merge.PlanMerging where
2
3  import qualified Lenses as L
4  import Types
5
6  import Control.Lens
7  import qualified Data.Map as M
8  import qualified Data.Map.Merge.Lazy as Merge
9
10 -----
11 --                                     Create Merge Plan
12 --
13 -- ↪ --

```

```

12 -----
13
14 createMergePlan ::
15     FlatModificationEvolutionPlan ->
16     FlatModificationEvolutionPlan ->
17     FlatModificationEvolutionPlan ->
18     MergeEvolutionPlan FlatFeatureModel
19 createMergePlan base v1 v2 =
20     base & L.plans
21     %~ \basePlans -> mergePlans basePlans (v1 ^. L.plans) (v2
22         ↪ ^. L.plans)
23
24 mergePlans ::
25     [Plan Modifications] ->
26     [Plan Modifications] ->
27     [Plan Modifications] ->
28     [Plan DiffResult]
29 mergePlans basePlans v1Plans v2Plans =
30     mergePlansWithTimes
31         (collectAllTimePoints basePlans v1Plans v2Plans)
32         basePlans
33         v1Plans
34         v2Plans
35
36 mergePlansWithTimes ::
37     [Time] ->
38     [Plan Modifications] ->
39     [Plan Modifications] ->
40     [Plan Modifications] ->
41     [Plan DiffResult]
42 mergePlansWithTimes [] _ _ _ = []
43 mergePlansWithTimes (time : times) basePlans v1Plans v2Plans =
44     Plan time (diffModifications baseModifications
45         ↪ v1Modifications v2Modifications) :
46     mergePlansWithTimes
47         times
48         nextBasePlans
49         nextV1Plans
50         nextV2Plans
51
52 where
53     (baseModifications, nextBasePlans) = getModificationForTime
54         ↪ basePlans time
55     (v1Modifications, nextV1Plans) = getModificationForTime
56         ↪ v1Plans time

```

```

52     (v2Modifications, nextV2Plans) = getModificationForTime
    ↪ v2Plans time
53
54 collectAllTimePoints :: [Plan a] -> [Plan a] -> [Plan a] ->
    ↪ [Time]
55 collectAllTimePoints basePlans v1Plans v2Plans =
56     merge (merge baseTimes v1Times) v2Times
57     where
58         baseTimes = basePlans ^.. traversed . L.timePoint
59         v1Times = v1Plans ^.. traversed . L.timePoint
60         v2Times = v2Plans ^.. traversed . L.timePoint
61         merge (x : xs) (y : ys)
62             | x == y = x : merge xs ys
63             | x < y = x : merge xs (y : ys)
64             | otherwise = y : merge (x : xs) ys
65         merge xs ys = xs ++ ys
66
67 getModificationForTime :: [Plan Modifications] -> Time ->
    ↪ (Modifications, [Plan Modifications])
68 getModificationForTime [] _ = (emptyModifications, [])
69 getModificationForTime plans@(Plan planTime modification :
    ↪ rest) time =
70     if time == planTime
71     then (modification, rest)
72     else (emptyModifications, plans)
73
74 emptyModifications :: Modifications
75 emptyModifications = Modifications M.empty M.empty
76
77 -- diffModifications will compare the modifcations from base
    ↪ with the
78 -- modifications from each derived version. The comparison
    ↪ will produce
79 -- a DiffResult that represents how every feature- and group
    ↪ modification was
80 -- changed between the base and derived versions
81 diffModifications :: Modifications -> Modifications ->
    ↪ Modifications -> DiffResult
82 diffModifications base v1 v2 =
83     DiffResult
84         (mergeMaps (base ^. L.features) (v1 ^. L.features) (v2 ^.
    ↪ L.features))
85         (mergeMaps (base ^. L.groups) (v1 ^. L.groups) (v2 ^.
    ↪ L.groups))
86     where

```

```

87     mergeMaps baseMap v1Map v2Map =
88         mergeBaseAndDerived
89         baseMap
90         $ mergeDerived v1Map v2Map
91
92 mergeBaseAndDerived ::
93     (Ord a, Eq modification) =>
94     M.Map a modification ->
95     M.Map a (DerivedComparisonResult modification) ->
96     M.Map a (SingleDiffResult modification)
97 mergeBaseAndDerived =
98     Merge.merge
99     ( Merge.mapMissing
100         (\_ baseMod -> withBase baseMod Nothing Nothing)
101     )
102     ( Merge.mapMissing
103         ( \_ derivedResult -> case derivedResult of
104             OneVersion version mod ->
105                 ChangedInOne
106                     version
107                     (OneChangeWithoutBase (AddedModification mod))
108             BothVersions v1Mod v2Mod ->
109                 ChangedInBoth
110                     ( BothChangeWithoutBase
111                         (AddedModification v1Mod)
112                         (AddedModification v2Mod)
113                     )
114         )
115     )
116     ( Merge.zipWithMatched
117         ( \_ baseMod derivedResult ->
118             case derivedResult of
119                 OneVersion V1 mod ->
120                     withBase baseMod (Just mod) Nothing
121                 OneVersion V2 mod ->
122                     withBase baseMod Nothing (Just mod)
123                 BothVersions v1Mod v2Mod ->
124                     withBase baseMod (Just v1Mod) (Just v2Mod)
125         )
126     )
127 where
128     withBase baseMod mV1Mod mV2Mod =
129         case (Just baseMod /= mV1Mod, Just baseMod /= mV2Mod) of
130             (True, True) ->
131                 ChangedInBoth

```



```

132         ( BothChangeWithBase
133           baseMod
134           (removeOrChanged mV1Mod)
135           (removeOrChanged mV2Mod)
136         )
137     (True, False) ->
138     ChangedInOne
139     V1
140     ( OneChangeWithBase
141       baseMod
142       (removeOrChanged mV1Mod)
143     )
144     (False, True) ->
145     ChangedInOne
146     V2
147     ( OneChangeWithBase
148       baseMod
149       (removeOrChanged mV2Mod)
150     )
151     (False, False) -> NoChange baseMod
152     removeOrChanged Nothing = RemovedModification
153     removeOrChanged (Just mod) = ChangedModification mod
154
155 data DerivedComparisionResult modification
156   = OneVersion Version modification
157   | BothVersions modification modification
158
159 mergeDerived ::
160   Ord a =>
161   M.Map a modification ->
162   M.Map a modification ->
163   M.Map a (DerivedComparisionResult modification)
164 mergeDerived =
165   Merge.merge
166     (Merge.mapMissing (const (OneVersion V1)))
167     (Merge.mapMissing (const (OneVersion V2)))
168     (Merge.zipWithMatched (const BothVersions))
169
170 -----
171 --                                     Unify Merge Plan
172 --                                     ↪  --
173 -----
174 unifyMergePlan ::
175   MergeEvolutionPlan FlatFeatureModel ->

```

```

176     Either Conflict FlatModificationEvolutionPlan
177 unifyMergePlan =
178     L.plans . traversed %%~ unifyTimePointResult
179
180 unifyTimePointResult ::
181     Plan DiffResult ->
182     Either Conflict (Plan Modifications)
183 unifyTimePointResult (Plan timePoint (DiffResult features
184     ↪ groups)) = do
185     features' <- unifyModificationsMap FeatureConflict timePoint
186     ↪ features
187     groups' <- unifyModificationsMap GroupConflict timePoint
188     ↪ groups
189     return $ Plan timePoint (Modifications features' groups')
190
191 unifyModificationsMap ::
192     Eq modificationType =>
193     (modificationIdType -> BothChange modificationType ->
194     ↪ MergeConflict) ->
195     Time ->
196     M.Map modificationIdType (SingleDiffResult modificationType)
197     ↪ ->
198     Either Conflict (M.Map modificationIdType modificationType)
199 unifyModificationsMap checkBothOverlapping timePoint =
200     M.traverseMaybeWithKey (unifySingleDiffResult
201     ↪ checkBothOverlapping timePoint)
202
203 unifySingleDiffResult ::
204     Eq modificationType =>
205     (modificationIdType -> BothChange modificationType ->
206     ↪ MergeConflict) ->
207     Time ->
208     modificationIdType ->
209     SingleDiffResult modificationType ->
210     Either Conflict (Maybe modificationType)
211 unifySingleDiffResult overlappingToMergeConflict timePoint id
212     ↪ singleDiffResult =
213     case singleDiffResult of
214     NoChange baseModification ->
215     Right (Just baseModification)
216     ChangedInOne version (OneChangeWithBase baseModification
217     ↪ RemovedModification) ->
218     Right Nothing
219     ChangedInOne version (OneChangeWithBase baseModification
220     ↪ (ChangedModification derivedModification)) ->

```

```

211     Right (Just derivedModification)
212 ChangedInOne version (OneChangeWithoutBase
    ↳ (AddedModification derivedModification)) ->
213     Right (Just derivedModification)
214 ChangedInBoth bothChange ->
215     checkOverlappingChanges overlappingToMergeConflict
    ↳ timePoint id bothChange
216
217 checkOverlappingChanges ::
218     Eq modificationType =>
219     (modificationIdType -> BothChange modificationType ->
    ↳ MergeConflict) ->
220     Time ->
221     modificationIdType ->
222     BothChange modificationType ->
223     Either Conflict (Maybe modificationType)
224 checkOverlappingChanges overlappingToMergeConflict timePoint id
    ↳ bothChange =
225     case bothChange of
226         BothChangeWithoutBase (AddedModification v1)
    ↳ (AddedModification v2) ->
227         ensureNotConflicting v1 v2
228         BothChangeWithBase base RemovedModification
    ↳ RemovedModification ->
229         Right Nothing
230         BothChangeWithBase base (ChangedModification v1)
    ↳ (ChangedModification v2) ->
231         ensureNotConflicting v1 v2
232         BothChangeWithBase{} ->
233         conflict
234     where
235         conflict = Left (Merge timePoint
    ↳ (overlappingToMergeConflict id bothChange))
236         ensureNotConflicting v1Modification v2Modification =
237             if v1Modification == v2Modification
238                 then Right (Just v1Modification)
239                 else conflict

```

## B.3 Check Plan

```

1 module Merge.CheckPlan where
2
3 import qualified Lenses as L

```

```

4 import Types
5
6 import Control.Lens
7 import Control.Monad.Error.Class
8 import Control.Monad.Writer.Lazy
9 import qualified Data.Map as M
10 import qualified Data.Set as S
11
12 -----
13 --                                Integrate All Modifications
14 --                                -----
15
16 integrateAndCheckModifications :: FlatModificationEvolutionPlan
17   ⇨ -> Either Conflict FlatUserEvolutionPlan
18 integrateAndCheckModifications evolutionPlan = case
19   ⇨ evolutionPlan of
20     TransformationEvolutionPlan initialTime initialFM plans ->
21       UserEvolutionPlan <$> scanEvolutionPlan plans (TimePoint
22         ⇨ initialTime initialFM)
23
24 scanEvolutionPlan ::
25   [Plan Modifications] -> TimePoint FlatFeatureModel -> Either
26   ⇨ Conflict [TimePoint FlatFeatureModel]
27 scanEvolutionPlan [] timePoint =
28   return [timePoint]
29 scanEvolutionPlan (plan : plans) currentTimePoint = do
30   (nextTimePointUnchecked, dependencies) <- runWriterT $
31     ⇨ integrateSinglePlan plan currentTimePoint
32   nextTimePoint <- checkGlobalConflict dependencies
33     ⇨ nextTimePointUnchecked
34   convertedEvolutionPlan <- scanEvolutionPlan plans
35     ⇨ nextTimePoint
36   return $ currentTimePoint : convertedEvolutionPlan
37
38 integrateSinglePlan ::
39   Plan Modifications ->
40   TimePoint FlatFeatureModel ->
41   WriterT [Dependency] (Either Conflict) (TimePoint
42     ⇨ FlatFeatureModel)
43 integrateSinglePlan (Plan nextTime modifications) (TimePoint
44   ⇨ prevTime featureModel) =
45   TimePoint nextTime <$> newFeatureModel
46   where

```

```

38     newFeatureModel = integrateFeatures featureModel >>=
    ↪ integrateGroups
39     integrateFeatures fm = ifoldlMOf (L.features . itraversed)
    ↪ (integrateFeature nextTime) fm modifications
40     integrateGroups fm = ifoldlMOf (L.groups . itraversed)
    ↪ (integrateGroup nextTime) fm modifications
41
42 integrateFeature ::
43     Time ->
44     FeatureId ->
45     FlatFeatureModel ->
46     FeatureModification ->
47     WriterT [Dependency] (Either Conflict) FlatFeatureModel
48 integrateFeature time featureId fm featureModification =
49     case featureModification of
50         FeatureAdd parentGroupId featureType name ->
51             case M.lookup featureId (fm ^. L.features) of
52                 Nothing -> do
53                     tell . fmap (FeatureDependency featureModification) $
54                         [ ParentGroupExists parentGroupId
55                           , UniqueName name
56                           , FeatureIsWellFormed featureId
57                         ]
58                     return $ fm & L.features . at featureId ?~
    ↪ FlatFeature (Just parentGroupId) featureType name
59                 Just oldFeature ->
60                     throwError $ Local time (FeatureAlreadyExists
    ↪ featureModification featureId)
61         FeatureRemove ->
62             case M.lookup featureId (fm ^. L.features) of
63                 Nothing ->
64                     throwError $ Local time (FeatureNotExists
    ↪ featureModification featureId)
65                 Just oldFeature -> do
66                     tell . fmap (FeatureDependency featureModification) $
67                         [NoChildGroups featureId]
68                     return $ fm & L.features . at featureId .~ Nothing
69 FeatureModification parentGroupIdMod featureTypeMod nameMod
    ↪ ->
70     if has (L.features . ix featureId) fm
71     then
72         pure fm
73         >>= integrateParentMod
74         >>= integrateTypeMod
75         >>= integrateNameMod

```

```

76     else
77         throwError $
78         Local time (FeatureNotExists featureModification
79             ↪ featureId)
80     where
81     integrateParentMod :: FlatFeatureModel -> WriterT
82         ↪ [Dependency] (Either Conflict) FlatFeatureModel
83     integrateParentMod fm =
84         case parentGroupIdMod of
85             Nothing -> return fm
86             Just (FeatureParentModification newValue) -> do
87                 tell . fmap (FeatureDependency
88                     ↪ featureModification) $
89                     [ ParentGroupExists newValue
90                       , NoCycleFromFeature featureId
91                       , FeatureIsWellFormed featureId
92                     ]
93                 return $ fm & L.features . ix featureId .
94                     ↪ L.parentGroupId .~ Just newValue
95
96     integrateTypeMod :: FlatFeatureModel -> WriterT
97         ↪ [Dependency] (Either Conflict) FlatFeatureModel
98     integrateTypeMod fm =
99         case featureTypeMod of
100             Nothing -> return fm
101             Just (FeatureTypeModification newValue) -> do
102                 tell . fmap (FeatureDependency
103                     ↪ featureModification) $
104                     [FeatureIsWellFormed featureId]
105                 return $ fm & L.features . ix featureId .
106                     ↪ L.featureType .~ newValue
107
108     integrateNameMod :: FlatFeatureModel -> WriterT
109         ↪ [Dependency] (Either Conflict) FlatFeatureModel
110     integrateNameMod fm =
111         case nameMod of
112             Nothing -> return fm
113             Just (FeatureNameModification newValue) -> do
114                 tell . fmap (FeatureDependency
115                     ↪ featureModification) $
116                     [UniqueName newValue]
117                 return $ fm & L.features . ix featureId . L.name
118                     ↪ .~ newValue
119
120 integrateGroup ::

```

```

111     Time ->
112     GroupId ->
113     FlatFeatureModel ->
114     GroupModification ->
115     WriterT [Dependency] (Either Conflict) FlatFeatureModel
116 integrateGroup time groupId fm groupModification =
117     case groupModification of
118     GroupAdd parentFeatureId groupType ->
119         case M.lookup groupId (fm ^. L.groups) of
120             Nothing -> do
121                 tell . fmap (GroupDependency groupModification) $
122                     [ ParentFeatureExists parentFeatureId
123                     ]
124                 return $ fm & L.groups . at groupId ?~ FlatGroup
125                     ↪ parentFeatureId groupType
126             Just oldGroup ->
127                 throwError $ Local time (GroupAlreadyExists
128                     ↪ groupModification groupId)
129     GroupRemove ->
130         case M.lookup groupId (fm ^. L.groups) of
131             Nothing ->
132                 throwError $ Local time (GroupNotExists
133                     ↪ groupModification groupId)
134             Just oldGroup -> do
135                 tell . fmap (GroupDependency groupModification) $
136                     [NoChildFeatures groupId]
137                 return $ fm & L.groups . at groupId .~ Nothing
138     GroupModification parentFeatureIdMod groupTypeMod ->
139         if has (L.groups . ix groupId) fm
140         then
141             pure fm
142             >>= integrateParentMod
143             >>= integrateTypeMod
144         else
145             throwError $
146                 Local time (GroupNotExists groupModification
147                     ↪ groupId)
148 where
149     integrateParentMod :: FlatFeatureModel -> WriterT
150     ↪ [Dependency] (Either Conflict) FlatFeatureModel
151     integrateParentMod fm =
152         case parentFeatureIdMod of
153             Nothing -> return fm
154             Just (GroupParentModification newValue) -> do
155                 tell . fmap (GroupDependency groupModification) $

```

```

151         [ ParentFeatureExists newValue
152         , NoCycleFromGroup groupId
153         ]
154     return $ fm & L.groups . ix groupId .
155         ↪ L.parentFeatureId .~ newValue
156
157 integrateTypeMod :: FlatFeatureModel -> WriterT
158     ↪ [Dependency] (Either Conflict) FlatFeatureModel
159 integrateTypeMod fm =
160     case groupTypeMod of
161     Nothing -> return fm
162     Just (GroupTypeModification newValue) -> do
163         tell . fmap (GroupDependency groupModification) $
164             [GroupIsWellFormed groupId]
165         return $ fm & L.groups . ix groupId . L.groupType
166             ↪ .~ newValue
167
168 checkGlobalConflict ::
169     [Dependency] ->
170     TimePoint FlatFeatureModel ->
171     Either Conflict (TimePoint FlatFeatureModel)
172 checkGlobalConflict dependencies tp@(TimePoint time
173     ↪ featureModel) =
174     errorIfFailed . filter (not . checkDependency) $ dependencies
175     where
176     errorIfFailed failedDeps =
177     case failedDeps of
178     [] -> Right tp
179     _ -> Left $ Global time (FailedDependencies failedDeps)
180 checkDependency (FeatureDependency featureMod
181     ↪ dependencyType) =
182     case dependencyType of
183     NoChildGroups featureId ->
184         hasn't
185         ( L.groups
186         . traversed
187         . L.parentFeatureId
188         . filtered (== featureId)
189         )
190         featureModel
191     ParentGroupExists groupId ->
192         has
193         (L.groups . ix groupId)
194         featureModel
195     NoCycleFromFeature featureId ->

```



```

191     not $ featureInCycle S.empty featureId featureModel
192 FeatureIsWellFormed featureId ->
193     -- If feature is mandatory, parent has to be AND
194     ↪ group
195     -- === feature not mandatory or parent is and
196     let featureType =
197         featureModel
198         ^?! L.features
199         . ix featureId
200         . L.featureType
201     parentGroupType =
202         featureModel
203         ^?! L.parentGroupOfFeature featureId
204         . L.groupType
205     in featureType /= Mandatory || parentGroupType ==
206     ↪ And
207 UniqueName name ->
208     lengthOf
209     (L.features . traversed . L.name . filtered (==
210     ↪ name))
211     featureModel
212     <= 1
213 checkDependency (GroupDependency groupMod dependencyType) =
214 case dependencyType of
215     NoChildFeatures groupId ->
216         hasn't
217         ( L.features
218         . traversed
219         . L.parentGroupId
220         . filtered (== Just groupId)
221         )
222         featureModel
223 ParentFeatureExists featureId ->
224     has
225     (L.features . ix featureId)
226     featureModel
227 NoCycleFromGroup groupId ->
228     not $ groupInCycle S.empty groupId featureModel
229 GroupIsWellFormed groupId ->
230     -- Either the group is a AND group, or all child
231     ↪ features are optional
232     let groupType = featureModel ^?! L.groups . ix
233     ↪ groupId . L.groupType
234     childFeatureTypes =
235         featureModel

```

```

231         ^.. L.childFeaturesOfGroup groupId
232         . L.featureType
233     in groupId == And || all (== Optional)
234         ⇨ childFeatureTypes
235
236 featureInCycle ::
237     S.Set (Either FeatureId GroupId) ->
238     FeatureId ->
239     FlatFeatureModel ->
240     Bool
241 featureInCycle visited featureId featureModel
242 | Left featureId `elem` visited = True
243 | otherwise =
244     case featureModel
245     ^? L.features
246     . ix featureId
247     . L.parentGroupId
248     . _Just of
249     Nothing -> False -- no parent group OR non existing
250     ⇨ feature
251     Just parentGroupId ->
252     groupInCycle
253     (S.insert (Left featureId) visited)
254     parentGroupId
255     featureModel
256
257 groupInCycle ::
258     S.Set (Either FeatureId GroupId) ->
259     GroupId ->
260     FlatFeatureModel ->
261     Bool
262 groupInCycle visited groupId featureModel
263 | Right groupId `elem` visited = True
264 | otherwise =
265     case featureModel
266     ^? L.groups
267     . ix groupId
268     . L.parentFeatureId of
269     Nothing -> False -- non existing group
270     Just parentFeatureId ->
271     featureInCycle
272     (S.insert (Right groupId) visited)
273     parentFeatureId
274     featureModel

```

```

274 -----
275 --                               Unflatten Evolution Plan
276 -----
277
278 unflattenSoundEvolutionPlan ::
279     FlatUserEvolutionPlan ->
280     TreeUserEvolutionPlan
281 unflattenSoundEvolutionPlan =
282     L.timePoints
283     . traversed
284     %~ unflattenTimePoint
285
286 unflattenTimePoint :: TimePoint FlatFeatureModel -> TimePoint
287     ⇨ TreeFeatureModel
288 unflattenTimePoint (TimePoint time featureModel) =
289     TimePoint time $
290     TreeFeatureModel $
291     unflattenFeature featureModel (featureModel ^. L.rootId)
292
293 unflattenFeature :: FlatFeatureModel -> FeatureId ->
294     ⇨ TreeFeature
295 unflattenFeature featureModel featureId =
296     TreeFeature featureId featureType name childGroups
297     where
298     childGroupIds = featureModel ^.. L.ichildGroupsOfFeature
299     ⇨ featureId . asIndex
300     childGroups = S.fromList $ fmap (unflattenGroup
301     ⇨ featureModel) childGroupIds
302     (FlatFeature _ featureType name) = featureModel ^?!
303     ⇨ L.features . ix featureId
304
305 unflattenGroup :: FlatFeatureModel -> GroupId -> TreeGroup
306 unflattenGroup featureModel groupId =
307     TreeGroup groupId groupType $ childFeatures
308     where
309     childFeatureIds = featureModel ^.. L.ichildFeaturesOfGroup
310     ⇨ groupId . asIndex
311     childFeatures = S.fromList $ fmap (unflattenFeature
312     ⇨ featureModel) childFeatureIds
313     (FlatGroup _ groupType) = featureModel ^?! L.groups . ix
314     ⇨ groupId

```