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

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[1].

[1] 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 **??**.

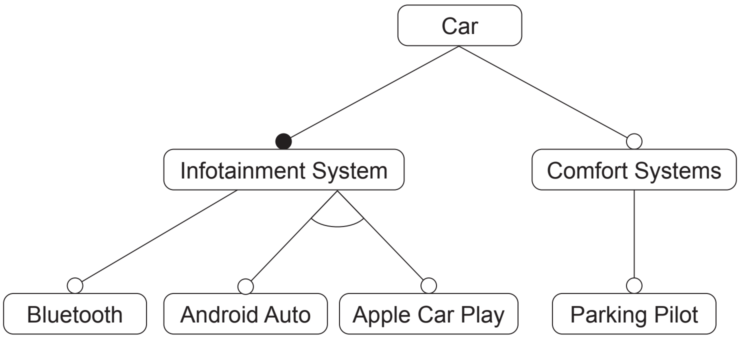


Figure 2.1: Example feature model

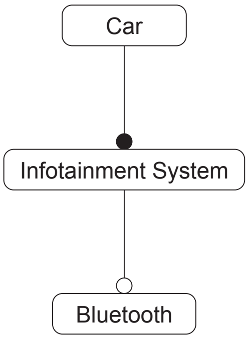
The small dot above `Infotainment System` indicates that the feature is mandatory, where as the white dot above `Comfort Systems` represents an optional feature. Each feature (except the root) is in a group. The `Infotainment System` feature is in a singleton group below `Car`. The features `Android 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 **??**.

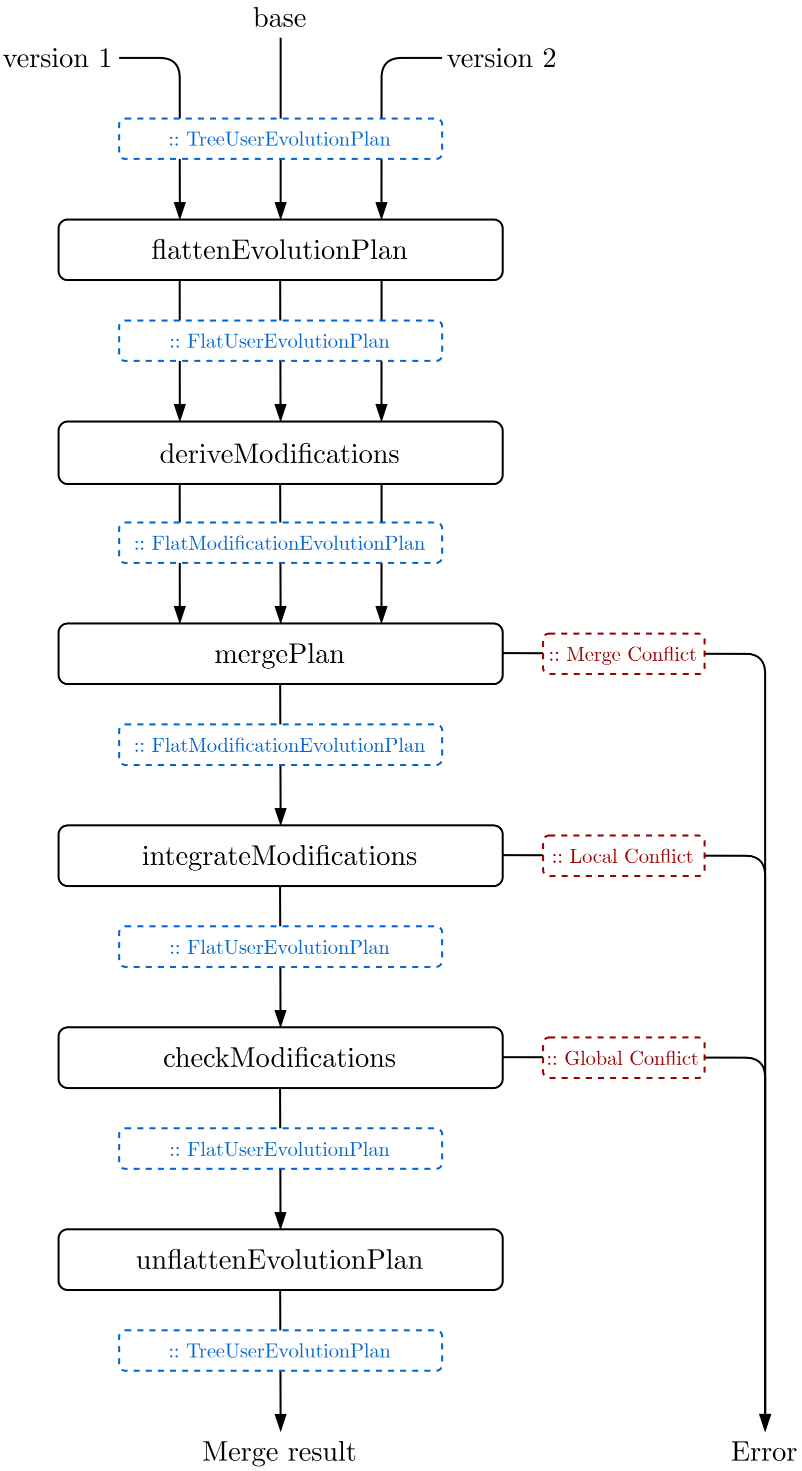


Figure 4.1: Outline of the three-way merge algorithm

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

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DerivingVia #-}
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}

module Types where

import qualified Data.Map as M
import qualified Data.Set as S

import Data.Aeson
import Deriving.Aeson
import Deriving.Aeson.Stock
import GHC.Generics

--------------------------------------------------------------------------
--                             Feature Models
 ↪  --
--------------------------------------------------------------------------

type FeatureId = String

type GroupId = String

--- Tree Structured Feature Model ---

data TreeFeatureModel = TreeFeatureModel
```

```haskell
  { _rootFeature :: TreeFeature
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" TreeFeatureModel

data TreeFeature = TreeFeature
  { _id :: FeatureId
  , _featureType :: FeatureType
  , _name :: String
  , _groups :: S.Set TreeGroup
  }
  deriving (Show, Eq, Read, Ord, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" TreeFeature

data TreeGroup = TreeGroup
  { _id :: GroupId
  , _groupType :: GroupType
  , _features :: S.Set TreeFeature
  }
  deriving (Show, Eq, Read, Ord, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" TreeGroup

--- Flat Structured Feature Model ---

data FlatFeatureModel = FlatFeatureModel
  { _rootId :: FeatureId
  , _features :: M.Map FeatureId FlatFeature
  , _groups :: M.Map GroupId FlatGroup
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" FlatFeatureModel

data FlatFeature = FlatFeature
  { _parentGroupId :: Maybe GroupId
  , _featureType :: FeatureType
  , _name :: String
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" FlatFeature

data FlatGroup = FlatGroup
  { _parentFeatureId :: FeatureId
  , _groupType :: GroupType
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" FlatGroup

data FeatureType
  = Optional
  | Mandatory
  deriving (Show, Eq, Read, Ord, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" FeatureType

data GroupType
  = And
  | Or
  | Alternative
  deriving (Show, Eq, Read, Ord, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" GroupType

--------------------------------------------------------------------------
--                          Evolution Plans
 ↪  --
--------------------------------------------------------------------------

--  Four different types of evolution plan representations.
 ↪  We categorize them in
--  two categories. User evolution plans and Tranformation
 ↪  evolution plans
--
--    User Evolution Plans:
--      Represents the evolution plan as a list of feature
 ↪  models, where each
```

```
--  feature model is coupled with a time point. In this representation the
--  exact changes between each feature model is implicit as the difference
--  between each pair of feature models
-- Transformation Evolution Plans:
--  Represents the evolution plan as an initial model, together with a list
--  of plans, where each plan is a time point and a transformation. The
--  transformation describes how the previous feature model should be
--  transformed in order to achieve the feature model at the given time
--  point. We define two different types of transformations, namely
--  Modification level and merge level modifications.
--
--  Modification  Transformation:
--   Represents the transformation as a set of modifications. This
--   representation guarantees that each there are no conflicting
--   modifications, i.e. moving a feature twice. This allows for merging
--   the modifications in an arbitrary ordering, since no modifications
--   shadow others, etc.
--  Merge  Transformation:
--   The merge level transformation represents the "planned"
--   transformations from both versions in the merge. The transformation
--   is essentially the union of the modifications of version 1 and
--   version 2. In this representation, a feature might be planned to be
--   changed, added or removed in several versions, which this
--   representation encodes.

type TreeUserEvolutionPlan = UserEvolutionPlan TreeFeatureModel

type FlatUserEvolutionPlan = UserEvolutionPlan FlatFeatureModel

```

```haskell
type FlatModificationEvolutionPlan = ModificationEvolutionPlan
 ↪  FlatFeatureModel

type Time = Int

data UserEvolutionPlan featureModel = UserEvolutionPlan
  { _timePoints :: [TimePoint featureModel]
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (UserEvolutionPlan featureModel)

data TimePoint featureModel = TimePoint
  { _time :: Time
  , _featureModel :: featureModel
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (TimePoint featureModel)

data TransformationEvolutionPlan transformation featureModel =
 ↪  TransformationEvolutionPlan
  { _initialTime :: Time
  , _initialFM :: featureModel
  , _plans :: [Plan transformation]
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (TransformationEvolutionPlan
     ↪  transformation featureModel)

data Plan transformation = Plan
  { _timePoint :: Time
  , _transformation :: transformation
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (Plan transformation)

type ModificationEvolutionPlan featureModel =
 ↪  TransformationEvolutionPlan Modifications featureModel
```

```haskell

type MergeEvolutionPlan featureModel = TransformationEvolutionPlan DiffResult featureModel

--------------------------------------------------------------------------
--                         Transformation Types                          --
--------------------------------------------------------------------------

--- MODIFICATIONS ---

-- Modifications vs Changes
-- We have two levels of changes. To differentiate between the two, we will use
-- the name Modification or Change in order to separate the two
--
-- Modifications:
--   Modifications are the actual changes between two feature models. For
--   example, If a feature was removed or added, we will call this "change" as
--   a Modification
--
-- Changes:
--   Changes are relevant to the diff-algorithm and its output, and refer to
--   the meta-level changes on modifications.  If a base version included
--   a Modification, i.e. an addition of a feature, one of the derived versions
--   could remove this modification The derived version has then Changed
--   a modification. So Change-names is reserved for these meta-level changes

--- Modifications between featuremodels ---

data Modifications = Modifications
  { _features :: M.Map FeatureId FeatureModification
  , _groups :: M.Map GroupId GroupModification
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" Modifications

data FeatureModification
  = FeatureAdd GroupId FeatureType String
  | FeatureRemove
  | FeatureModification
      (Maybe FeatureParentModification)
      (Maybe FeatureTypeModification)
      (Maybe FeatureNameModification)
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" FeatureModification

data FeatureParentModification
  = FeatureParentModification GroupId
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" FeatureParentModification

data FeatureNameModification
  = FeatureNameModification String
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" FeatureNameModification

data FeatureTypeModification
  = FeatureTypeModification FeatureType
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" FeatureTypeModification

data GroupModification
  = GroupAdd FeatureId GroupType
  | GroupRemove
  | GroupModification
      (Maybe GroupParentModification)
      (Maybe GroupTypeModification)
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" GroupModification
```

```haskell

data GroupParentModification
  = GroupParentModification FeatureId
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" GroupParentModification

data GroupTypeModification
  = GroupTypeModification GroupType
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" GroupTypeModification

--- DIFF RESULT ---

-- The diff result from the all the changes in the entire time point for all
-- versions of the model
data DiffResult = DiffResult
  { _features :: M.Map FeatureId FeatureDiffResult
  , _groups :: M.Map GroupId GroupDiffResult
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" DiffResult

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)
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (SingleDiffResult modificationType)
```

```haskell

data OneChange modificationType
  = OneChangeWithBase
      modificationType -- Base modification
      (RemovedOrChangedModification modificationType) -- Derived (V1 or V2) modification
  | OneChangeWithoutBase
      (AddedModification modificationType) -- Derived (V1 or V2) modification
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (OneChange modificationType)

data BothChange modificationType
  = BothChangeWithBase
      modificationType -- Base modification
      (RemovedOrChangedModification modificationType) -- V1 modification
      (RemovedOrChangedModification modificationType) -- V2 modification
  | BothChangeWithoutBase
      (AddedModification modificationType) -- V1 modification
      (AddedModification modificationType) -- V2 modification
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (BothChange modificationType)

data RemovedOrChangedModification modificationType
  = RemovedModification
  | ChangedModification modificationType
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (RemovedOrChangedModification modificationType)

data AddedModification modificationType
  = AddedModification modificationType
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (AddedModification modificationType)

```

```haskell
data Version
  = V1
  | V2
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" Version

-------------------------------------------------------------------------
--                       Merge Input / Output                          --
-------------------------------------------------------------------------

data MergeInput
  = TreeUser (MergeInputData TreeUserEvolutionPlan)
  | FlatUser (MergeInputData FlatUserEvolutionPlan)
  | FlatModification (MergeInputData FlatModificationEvolutionPlan)
  deriving (Show, Eq, Read)

data MergeInputData evolutionPlan = MergeInputData
  { _name :: String
  , _base :: evolutionPlan
  , _v1 :: evolutionPlan
  , _v2 :: evolutionPlan
  , _maybeExpected :: Maybe (MergeResult evolutionPlan)
  }
  deriving (Show, Eq, Read, Generic, Functor)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" (MergeInputData evolutionPlan)

type MergeOutput = Either Conflict (FlatModificationEvolutionPlan, FlatUserEvolutionPlan)

type MergeResult evolutionPlan = Either Conflict evolutionPlan

-------------------------------------------------------------------------
--                       Elm Data Serialization                        --
-------------------------------------------------------------------------

data ElmDataExamples = ElmDataExamples
  { _examples :: [ElmMergeExample]
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" ElmDataExamples

data ElmMergeExample = ElmMergeExample
  { _name :: String
  , _evolutionPlans :: [ElmNamedEvolutionPlan]
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" ElmMergeExample

data ElmNamedEvolutionPlan = ElmNamedEvolutionPlan
  { _name :: String
  , _mergeData :: Either String TreeUserEvolutionPlan
  }
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" ElmNamedEvolutionPlan

-------------------------------------------------------------------------
--                               Conflict                              --
-------------------------------------------------------------------------

data Conflict
  = Merge Time MergeConflict
  | Local Time LocalConflict
  | Global Time GlobalConflict
  | Panic Time String
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" Conflict

data MergeConflict
  = FeatureConflict FeatureId (BothChange FeatureModification)
  | GroupConflict GroupId (BothChange GroupModification)
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" MergeConflict
```

```haskell

data LocalConflict
  = FeatureAlreadyExists FeatureModification FeatureId
  | FeatureNotExists FeatureModification FeatureId
  | GroupAlreadyExists GroupModification GroupId
  | GroupNotExists GroupModification GroupId
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" LocalConflict

data GlobalConflict
  = FailedDependencies [Dependency]
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" GlobalConflict

data Dependency
  = FeatureDependency FeatureModification FeatureDependencyType
  | GroupDependency GroupModification GroupDependencyType
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" Dependency

data FeatureDependencyType
  = NoChildGroups FeatureId
  | ParentGroupExists GroupId
  | NoCycleFromFeature FeatureId
  | FeatureIsWellFormed FeatureId
  | UniqueName String
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" FeatureDependencyType

data GroupDependencyType
  = NoChildFeatures GroupId
  | ParentFeatureExists FeatureId
  | NoCycleFromGroup GroupId
  | GroupIsWellFormed GroupId
  deriving (Show, Eq, Read, Generic)
  deriving
    (FromJSON, ToJSON)
    via Prefixed "_" GroupDependencyType

--------------------------------------------------------------------------
--                             CLI OPTIONS                              --
--------------------------------------------------------------------------

data EvolutionPlanType
  = TreeUserType
  | FlatUserType
  | FlatModificationType
  deriving (Show, Eq, Read)

data Mode
  = GenerateOne String
  | GenerateAll
  | FromFile FilePath
  deriving (Show, Eq, Read)

data CliOptions = CliOptions
  { _mode :: Mode
  , _fromType :: EvolutionPlanType
  , _toType :: EvolutionPlanType
  , _print :: Bool
  , _generateElm :: Bool
  , _toFile :: Maybe FilePath
  }
  deriving (Show, Eq, Read)
```

Appendix B

Three Way Merge Algorithm

```haskell
{-# LANGUAGE FlexibleContexts #-}

module ThreeWayMerge where

import Convertable
import Merge.CheckPlan (integrateAndCheckModifications)
import Merge.PlanMerging (createMergePlan, unifyMergePlan)
import Types

threeWayMerge ::
  ConvertableInput inputEvolutionPlan =>
  MergeInputData inputEvolutionPlan ->
  MergeOutput
threeWayMerge (MergeInputData _ base v1 v2 _) = do
  let mergePlan =
        createMergePlan
          (toFlatModification base)
          (toFlatModification v1)
          (toFlatModification v2)
  mergedModificationPlan <- unifyMergePlan mergePlan
  checkedUserFlatPlan <- integrateAndCheckModifications mergedModificationPlan
  return (mergedModificationPlan, checkedUserFlatPlan)
```

B.1 Change Detection

```haskell
module Merge.ChangeDetection where

```

```haskell
import qualified Lenses as L
import Types

import Control.Lens
import qualified Data.Map as M
import qualified Data.Map.Merge.Lazy as Merge

--------------------------------------------------------------------------
--                    Flatten Sound Evolution Plan                      --
--------------------------------------------------------------------------

flattenSoundEvolutionPlan :: TreeUserEvolutionPlan -> FlatUserEvolutionPlan
flattenSoundEvolutionPlan =
  L.timePoints
    . traversed
    . L.featureModel
    %~ flattenSoundFeatureModel

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 mParentGroup (TreeFeature id featureType name groups) =
      ([(id, FlatFeature mParentGroup featureType name)], [])
        <> foldMap (flattenGroup id) groups
    flattenGroup parentFeature (TreeGroup id groupType features) =
      ([], [(id, FlatGroup parentFeature groupType)])
        <> foldMap (flattenFeature (Just id)) features

--------------------------------------------------------------------------
--                     Derive Sound Modifications                       --
--------------------------------------------------------------------------

```

```haskell
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

-- diffFeatureModels will derive every modification
diffFeatureModels :: FlatFeatureModel -> FlatFeatureModel -> Modifications
diffFeatureModels prevFM currFM =
  Modifications
    featureModifications
    groupModifications
  where
    featureModifications =
      Merge.merge
        (Merge.mapMissing (\_ _ -> FeatureRemove))
        ( Merge.mapMissing
            ( \_ (FlatFeature mParent featureType name) ->
                case mParent of
                  Nothing ->
                    error $
                      "ERROR: When diffing two feature models, "
                        ++ "the root feature is assumed to be the same in both version. "
                        ++ "Since the root feature cannot be removed, there should never "
                        ++ "be the case that the root feature was added"
                  Just parent -> FeatureAdd parent featureType name
            )
        )
        ( Merge.zipWithMaybeMatched
            ( \_
               prev@(FlatFeature prevParent prevFeatureType
                ↪  prevName)
               new@(FlatFeature newParent newFeatureType
                ↪  newName) ->
                  if prev == new
                    then Nothing
                    else
                      Just $
                        FeatureModification
                          ( case (prevParent, newParent) of
                              (Just prev, Just new) | prev /=
                                  ↪  new -> Just
                                  ↪  (FeatureParentModification
                                  ↪  new)
                              -- NOTE: since the root is
                               ↪  assumed to never change,
                              -- we only record changes of
                               ↪  non-root features
                              _ -> Nothing
                          )
                          ( if prevFeatureType ==
                           ↪  newFeatureType
                              then Nothing
                              else Just
                                  ↪  (FeatureTypeModification
                                  ↪  newFeatureType)
                          )
                          ( if prevName == newName
                              then Nothing
                              else Just
                                  ↪  (FeatureNameModification
                                  ↪  newName)
                          )
            )
        )
        (prevFM ^. L.features)
        (currFM ^. L.features)
    groupModifications =
      Merge.merge
        (Merge.mapMissing (\_ _ -> GroupRemove))
        ( Merge.mapMissing
            ( \_ (FlatGroup parent groupType) ->
                GroupAdd parent groupType
            )
        )
        ( Merge.zipWithMaybeMatched
            ( \_
               prev@(FlatGroup prevParent prevGroupType)
               new@(FlatGroup newParent newGroupType) ->
                  if prev == new
                    then Nothing
                    else
                      Just $
                        GroupModification
                          ( if prevParent == newParent
                              then Nothing
                              else Just
                               ↪  (GroupParentModification
                               ↪  newParent)
                          )
                          ( if prevGroupType == newGroupType
                              then Nothing
                              else Just (GroupTypeModification
                               ↪  newGroupType)
                          )
            )
        )
        (prevFM ^. L.groups)
        (currFM ^. L.groups)
```

B.2 Plan Merging

```
module Merge.PlanMerging where

import qualified Lenses as L
import Types

import Control.Lens
import qualified Data.Map as M
import qualified Data.Map.Merge.Lazy as Merge

--------------------------------------------------------------------------
--                          Create Merge Plan
 ↪  --
```

```
--------------------------------------------------------------------------

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
```

```haskell
    (v2Modifications, nextV2Plans) = getModificationForTime v2Plans time

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@(Plan planTime modification : rest) time =
  if time == planTime
    then (modification, rest)
    else (emptyModifications, plans)

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

-- diffModifications will compare the modifcations from base with the
-- modifications from each derived version. The comparison will produce
-- a DiffResult that represents how every feature- and group modification was
-- changed between the base and derived versions
diffModifications :: Modifications -> Modifications -> Modifications -> DiffResult
diffModifications base v1 v2 =
  DiffResult
    (mergeMaps (base ^. L.features) (v1 ^. L.features) (v2 ^. L.features))
    (mergeMaps (base ^. L.groups) (v1 ^. L.groups) (v2 ^. L.groups))
  where
```

```haskell
    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
        (\_ baseMod -> withBase baseMod Nothing Nothing)
    )
    ( Merge.mapMissing
        ( \_ derivedResult -> case derivedResult of
            OneVersion version mod ->
              ChangedInOne
                version
                (OneChangeWithoutBase (AddedModification mod))
            BothVersions v1Mod v2Mod ->
              ChangedInBoth
                ( BothChangeWithoutBase
                    (AddedModification v1Mod)
                    (AddedModification v2Mod)
                )
        )
    )
    ( Merge.zipWithMatched
        ( \_ baseMod derivedResult ->
            case derivedResult of
              OneVersion V1 mod ->
                withBase baseMod (Just mod) Nothing
              OneVersion V2 mod ->
                withBase baseMod Nothing (Just mod)
              BothVersions v1Mod v2Mod ->
                withBase baseMod (Just v1Mod) (Just v2Mod)
        )
    )
  where
    withBase baseMod mV1Mod mV2Mod =
      case (Just baseMod /= mV1Mod, Just baseMod /= mV2Mod) of
        (True, True) ->
          ChangedInBoth
```

```haskell
            ( 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))

--------------------------------------------------------------------------
--                           Unify Merge Plan                           --
--------------------------------------------------------------------------

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

unifyTimePointResult ::
  Plan DiffResult ->
  Either Conflict (Plan Modifications)
unifyTimePointResult (Plan timePoint (DiffResult features
 ↪  groups)) = do
  features' <- unifyModificationsMap FeatureConflict timePoint
   ↪  features
  groups' <- unifyModificationsMap GroupConflict timePoint
   ↪  groups
  return $ Plan timePoint (Modifications features' groups')

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

unifySingleDiffResult ::
  Eq modificationType =>
  (modificationIdType -> BothChange modificationType ->
   ↪  MergeConflict) ->
  Time ->
  modificationIdType ->
  SingleDiffResult modificationType ->
  Either Conflict (Maybe modificationType)
unifySingleDiffResult overlappingToMergeConflict timePoint id
 ↪  singleDiffResult =
  case singleDiffResult of
    NoChange baseModification ->
      Right (Just baseModification)
    ChangedInOne version (OneChangeWithBase baseModification
     ↪  RemovedModification) ->
      Right Nothing
    ChangedInOne version (OneChangeWithBase baseModification
     ↪  (ChangedModification derivedModification)) ->
```

```
      Right (Just derivedModification)
    ChangedInOne version (OneChangeWithoutBase (AddedModification derivedModification)) ->
      Right (Just derivedModification)
    ChangedInBoth bothChange ->
      checkOverlappingChanges overlappingToMergeConflict timePoint id bothChange

checkOverlappingChanges ::
  Eq modificationType =>
  (modificationIdType -> BothChange modificationType -> MergeConflict) ->
  Time ->
  modificationIdType ->
  BothChange modificationType ->
  Either Conflict (Maybe modificationType)
checkOverlappingChanges overlappingToMergeConflict timePoint 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 timePoint (overlappingToMergeConflict id bothChange))
    ensureNotConflicting v1Modification v2Modification =
      if v1Modification == v2Modification
        then Right (Just v1Modification)
        else conflict
```

B.3 Check Plan

```
module Merge.CheckPlan where

import qualified Lenses as L
```

```haskell
import Types

import Control.Lens
import Control.Monad.Error.Class
import Control.Monad.Writer.Lazy
import qualified Data.Map as M
import qualified Data.Set as S

------------------------------------------------------------------------------
--                     Integrate All Modifications                          --
------------------------------------------------------------------------------

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

integrateSinglePlan ::
  Plan Modifications ->
  TimePoint FlatFeatureModel ->
  WriterT [Dependency] (Either Conflict) (TimePoint FlatFeatureModel)
integrateSinglePlan (Plan nextTime modifications) (TimePoint prevTime featureModel) =
  TimePoint nextTime <$> newFeatureModel
  where
```

```haskell
    newFeatureModel = integrateFeatures featureModel >>= integrateGroups
    integrateFeatures fm = ifoldlMOf (L.features . itraversed) (integrateFeature nextTime) fm modifications
    integrateGroups fm = ifoldlMOf (L.groups . itraversed) (integrateGroup nextTime) fm modifications

integrateFeature ::
  Time ->
  FeatureId ->
  FlatFeatureModel ->
  FeatureModification ->
  WriterT [Dependency] (Either Conflict) FlatFeatureModel
integrateFeature time featureId fm featureModification =
  case featureModification of
    FeatureAdd parentGroupId featureType name ->
      case M.lookup featureId (fm ^. L.features) of
        Nothing -> do
          tell . fmap (FeatureDependency featureModification) $
            [ ParentGroupExists parentGroupId
            , UniqueName name
            , FeatureIsWellFormed featureId
            ]
          return $ fm & L.features . at featureId ?~ FlatFeature (Just parentGroupId) featureType name
        Just oldFeature ->
          throwError $ Local time (FeatureAlreadyExists featureModification featureId)
    FeatureRemove ->
      case M.lookup featureId (fm ^. L.features) of
        Nothing ->
          throwError $ Local time (FeatureNotExists featureModification featureId)
        Just oldFeature -> do
          tell . fmap (FeatureDependency featureModification) $
            [NoChildGroups featureId]
          return $ fm & L.features . at featureId .~ Nothing
    FeatureModification parentGroupIdMod featureTypeMod nameMod ->
      if has (L.features . ix featureId) fm
        then
          pure fm
            >>= integrateParentMod
            >>= integrateTypeMod
            >>= integrateNameMod
    else
     throwError $
      Local time (FeatureNotExists featureModification
      ↪ featureId)
   where
    integrateParentMod :: FlatFeatureModel -> WriterT
    ↪ [Dependency] (Either Conflict) FlatFeatureModel
    integrateParentMod fm =
     case parentGroupIdMod of
      Nothing -> return fm
      Just (FeatureParentModification newValue) -> do
       tell . fmap (FeatureDependency
       ↪ featureModification) $
        [ ParentGroupExists newValue
        , NoCycleFromFeature featureId
        , FeatureIsWellFormed featureId
        ]
       return $ fm & L.features . ix featureId .
       ↪ L.parentGroupId .~ Just newValue

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

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

integrateGroup ::
```

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

```
                [ ParentFeatureExists newValue
                , NoCycleFromGroup groupId
                ]
              return $ fm & L.groups . ix groupId . L.parentFeatureId .~ newValue

        integrateTypeMod :: FlatFeatureModel -> WriterT [Dependency] (Either Conflict) FlatFeatureModel
        integrateTypeMod fm =
          case groupTypeMod of
            Nothing -> return fm
            Just (GroupTypeModification newValue) -> do
              tell . fmap (GroupDependency groupModification) $
                [GroupIsWellFormed groupId]
              return $ fm & L.groups . ix groupId . L.groupType .~ newValue

checkGlobalConflict ::
  [Dependency] ->
  TimePoint FlatFeatureModel ->
  Either Conflict (TimePoint FlatFeatureModel)
checkGlobalConflict dependencies tp@(TimePoint time featureModel) =
  errorIfFailed . filter (not . checkDependency) $ dependencies
  where
    errorIfFailed failedDeps =
      case failedDeps of
        [] -> Right tp
        _ -> Left $ Global time (FailedDependencies failedDeps)
    checkDependency (FeatureDependency featureMod dependencyType) =
      case dependencyType 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 feature is mandatory, 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 dependencyType) =
      case dependencyType of
        NoChildFeatures groupId ->
          hasn't
            ( L.features
                . traversed
                . L.parentGroupId
                . filtered (== Just groupId)
            )
            featureModel
        ParentFeatureExists featureId ->
          has
            (L.features . ix featureId)
            featureModel
        NoCycleFromGroup groupId ->
          not $ groupInCycle S.empty groupId featureModel
        GroupIsWellFormed groupId ->
          -- Either the group is a AND group, or all child features are optional
          let groupType = featureModel ^?! L.groups . ix groupId . L.groupType
              childFeatureTypes =
                featureModel
```

```
                  ^.. L.childFeaturesOfGroup groupId
                    . L.featureType
           in groupType == And || all (== Optional) childFeatureTypes

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

```

```
--------------------------------------------------------------------------
--                      Unflatten Evolution Plan
 ↪  --
--------------------------------------------------------------------------

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
```