Master Thesis

Methods for modular soundness checker of feature model evolution plans

Ida Sandberg Motzfeldt

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
(Software)
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Master Thesis

Methods for modular soundness checker of feature model evolution plans

Ida Sandberg Motzfeldt

Abstract

Contents

List of Figures

List of Tables

Preface

Part I

Introduction

Giving the thesis a purpose (not how, but why) Mention problems in industry practice (mention software product lines) Say what software product lines are for, not technicalities, why they exist Talk about why planning is difficult, when the project size is increasing -¿ Changing the plan is even more difficult, and there is a lack of automated tools for checking that the changes don't break anything Drawing a picture of the domain and the problem My thesis is addressing these challenges by looking at these research questions or goals Last part of introduction: research questions/goals (base structure of thesis on this) - 3 is a good number They are general. What, how, (never why) In this thesis, I will address these questions through these activities ...(will not tackle everything, so make the purpose clear) Start broader, chunk down to more detail Can take the goals up again in the following sections (mention the research questions when tackling them) How the sensor should read my master's thesis. How will she know what my contribution is, where it comes in. "In chapter 2 I will give the background, limited to the scope of my thesis". Make it clear that I'm not trying to cover everything, and give the logic/structure of the thesis.

TODO: Reread essay and identify useful parts

Chapter 1

Background

Software product lines, previous publication (pinpoint what can be improved), SOS rules and operational semantics, and what is modular analysis Background should not contain many of my results, but rather retelling what software product lines are, what operational semantics are, existing technologies What is the difficulty of doing analysis on it? Just explaining the structures does not convey why it is worth spending time to solve the problems. Hint at what is coming, that this is something I will tackle in the contribution. Can't be technical in the introduction, the background is where I introduce the terms and definitions and technical details. And the implications of these details. Explain what I mean by "modularity", explain why the previous work is not modular. What do I assume, and what do I focus on

Part II

The project

Help the reader distinguish the plan and the change of plan (maybe in background). Make an illustration that can help convey the terminology (when I say this word, I'm in this level, etc.) Forskerlinjen is part of the story and research work. Can include what we did earlier, since the thesis is based on this work. Focus on the operations. Not necessary to have the rules here, but can refer to the paper. Can use it to explain what it means to be paradox-free. Can say what the plan looks like and what a sound plan is, and refer the reader to the paper for more details. Then say in this project I will look at modular modifications of a plan (the previous work does not tackle plan change). Since it is already published, the sensor does not need to evaluate that part, only the continuation which is this thesis. However, the previous work is a necessary part of the story of this thesis.

TODO: Need to structure the project part more finely

Chapter 2

Definitions and semantics

TODO: Explain why it's non-trivial, why it may be difficult to do it manually, why we need to restrict the scope etc. Why simple lookup is not enough.

2.1 Definitions

2.1.1 Temporal feature model

A temporal feature model is defined as a triple (NAMES, FEATURES, GROUPS) where NAMES, FEATURES, and GROUPS are all maps. The reason for this choice is efficiency and modularity. As mentioned in TODO: motivation or background or something, the goal of this thesis is to minimize the scope of the plan to check for paradoxes, as a change rarely affects more than a small part of the plan. It would then be a mistake to represent a plan as a sequence of trees associated with time points, or an initial model followed by a sequence of operations as described in "Consistency-preserving evolution planning on feature models" (Motzfeldt et al. 2020). To add a new feature to the plan, both representations would require us to look through the entire plan to check that the feature ID is fresh.

To add or rename a feature, a soundness checker must verify that no other feature is using the name during the affected part of the plan. We therefore include the NAMES map in the representation for efficient verification of aforementioned issue. A feature or group ID may not already be in use when we add it, so the FEATURES and GROUPS maps support efficient lookup for IDs. The rest of this section gives more detailed explanations of temporal feature models.

Maps

In general, a map is a set of entries on the form $[k \mapsto v]$, where each key k uniquely defines a value v. Looking up a value at the key k in map MAP looks like this:

$$\text{MAP}\,[k]$$

This query would give us v if $[\,k \mapsto v\,] \in \text{MAP}$. If we wish to assign a value v' to key k, this is the syntax:

$$\text{MAP}\,[k] \leftarrow v'$$

This also works if k is not already in the map. We assume $O(1)$ for lookup and insertion. For maps with set values, we define an additional operator $\overset{\cup}{\leftarrow}$. If $\text{MAP}\,[k] = S$ then

$$\text{MAP}\,[k] \overset{\cup}{\leftarrow} v = \text{MAP}\,[k] \leftarrow S \cup \{v\}$$

To remove a mapping with key k, we use $\text{MAP} \setminus k$. For maps with set values, we additionally define $\setminus^v$, where v is some value. Let MAP be a map with set values containing the mapping $[\,k \mapsto \{v\} \cup S\,]$. Then $\setminus^v$ is defined as follows:

$$\text{MAP} \setminus^v k = \begin{cases} \text{MAP} \setminus k & \text{if } S = \emptyset \\ \text{MAP}\,[k] \leftarrow S & \text{if } |S| > 0 \end{cases}$$

That is, if removing v leaves only the empty set at $\text{MAP}\,[k]$, we remove the mapping. Otherwise, we only remove v from the set of values associated with k.

Intervals

To define the map values, we must first define an interval. We use the familiar mathematical notation $[t_{\text{start}}, t_{\text{end}})$ for the intervals, where t_{start} and t_{end} denote points in time. The intervals are left-closed and right-open, meaning that t_{start} is included in the interval, and all time points until but not including t_{end}. The intervals are always used to give information about when something is true in the temporal feature model. This will be further explained in subsections 2.1.1, 2.1.1. Single time points t_n may be viewed as intervals containing only the time point t_n

For intervals $[t_{start}, t_{end})$ with unknown bounds, we may restrict the bounds to t_l and t_r by writing $\langle [t_{start}, t_{end}) \rangle_{t_l}^{t_r}$. We then get the interval $[max(t_{start}, t_l), min(t_{end}, t_r))$

Interval maps

An interval map is a map where the key is an interval (section 2.1.1). To look up values, one can either give an interval or a time point as key. Both will return sets of values. For instance, if an interval map I contains the mapping $[\,[t_1, t_5) \mapsto v\,]$, all of the queries in Figure 2.1 on the next page will return $\{v\}$ (assuming that $t_1 < t_2 < \ldots < t_5$):

$$
\begin{array}{l}
\text{I}\,[t_1] \\
\text{I}\,[t_3] \\
\text{I}\,[[t_1, t_5)] \\
\text{I}\,[[t_2, t_4)]
\end{array}
$$

Figure 2.1: Interval map example

IM $[t_n]_{\leq}$ returns the set of keys containing time point t_n. For interval maps with non-overlapping keys, the will contain at most one element. For interval maps with set values, we define an additional function IM $[t_n]_{\leq}^{v}$ where v is some value, returning the set of the keys containing t_n and associated with a set containing v.

We furthermore define function IM$[t_n, t_m)_{\lesseqgtr}$ which returns all the interval keys in the map IM overlapping the interval $[t_n, t_m)$. See section 2.1.1 for definition of overlapping intervals.

Interval sets

An interval set is a set of intervals with a few custom predicates and operations. Given an interval set IS, $[t_n, t_m) \in$ IS if $[t_n, t_m)$ is a member of the set, which is the expected semantics of $\in$. We define a similar predicate $\in_{\leq}$ such that $[t_n, t_m) \in_{\leq}$ IS iff there exists some interval $[t_i, t_j) \in$ IS with $t_i \leq t_n \leq t_m \leq t_j$, i.e. an interval in IS which contains $[t_n, t_m)$. We further define the predicate $\in_{\lesseqgtr}$ such that $[t_n, t_m) \in_{\lesseqgtr}$ IS iff there exists some interval $[t_i, t_j) \in$ IS with $[t_n, t_m)$ overlapping $[t_i, t_j)$. Two intervals $[t_n, t_m)$ and $[t_i, t_j)$ overlap iff there exists a time point t_k with $t_n \leq t_k < t_m$ and $t_i \leq t_k < t_j$, i.e. a time point contained by both intervals.

Notice that $\in \subseteq \in_{\leq} \subseteq \in_{\lesseqgtr}$, which means that if $[t_n, t_m) \in$ IS then also $[t_n, t_m) \in_{\leq}$ IS, and $[t_n, t_m) \in_{\lesseqgtr}$ IS.

IS $[t_n]_{\leq}$ returns the subset of IS containing t_n.

Mapping names

The NAMES map has entries of the form [**name** $\mapsto$ *interval map*]. Assuming [`name` $\mapsto$ IM] $\in$ NAMES, the interval map IM contains mappings on the form $[\,[t_{\text{start}}, t_{\text{end}}) \mapsto$ `featureID`], where `featureID` is the ID of some feature in the temporal feature model. This should be interpreted as "*The name* `name` *belongs to the feature with ID* `featureID` *from* t_{start} *to* t_{end}". Looking up a name which does not exist will return an empty map $\emptyset$.

This map is mainly used when adding features or changing names. The new name and the scope of the change is then looked up in the NAMES map to verify that no other feature shares the name.

Mapping features

The FEATURES map has entries of the form $[\texttt{featureID} \mapsto \textit{feature entry}]$. Since several pieces of information are crucial to the analysis of a feature, it is not enough to have a simple mapping as we have for names. A feature has a name, a type, a parent group, and zero or more child groups. Furthermore, a feature may be removed and re-added during the course of the plan, so we also need information about when the feature exists. This information is collected into a 5-tuple $(F_e, F_n, F_t, F_p, F_c)$, where F_e is an interval set denoting when the feature exists, F_n is an interval map with name values, F_t is an interval map with the feature's variation types, F_p is an interval map with group ID values, and F_c is an interval map where the values are sets containing group IDs, the interval keys possibly overlapping.

Looking up a feature which does not exist returns an empty feature $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$. This lets us treat an unsuccessful lookup the same way as a successful one.

The root ID is constant for a temporal feature model. We assume that it has been computed and is stored in the variable `rootID`.

The reasoning behind the choice of interval sets and maps here is in large part the temporal scope; for instance, when a feature is removed, we can easily look up the temporal scope in the F_c map (child groups) to verify that removing the feature leaves no group without a parent.

Mapping groups

The GROUPS map has entries of the form $[\texttt{groupID} \mapsto \textit{group entry}]$. A group has a type, a parent feature, and zero or more child features. It may also be removed and re-added These can all be defined in terms of intervals and collected into a 4-tuple similarly to the feature entries (G_e, G_t, G_p, G_c), where G_e is an interval set denoting when the group exists, G_t is a map with the group's types, G_p is a map with feature IDs, and G_c is a map with feature ID values, the interval keys possibly overlapping.

Looking up a group which does not exist in the map returns an empty group $(\emptyset, \emptyset, \emptyset, \emptyset)$.

Example evolution plan

A small example of a temporal feature model can be found in Figure 2.2 on page 11. It contains three features and one group, and describes a temporal feature model for a washing machine. The washing machine always has a washer, and a dryer is added at t_5. It is clear from this example that the representation is better suited for manipulating the structure than reading it.

2.1.2 Operations

We define *operations* to alter the temporal feature model. A software product line may grow very large, and the plans even larger. Since different factors may influence the plan, it is necessary to be able to change the plan accordingly. If the plan is indeed extremely large, and since feature models have strict structure constraints, it is also necessary to check *automatically* that the changes do not compromise the structure. Due to the size and complexity of the problem, it is not enough to let a human verify a change.

- **addFeature(featureID, parentGroupID, name, featureType)** from t_n to t_m
  Adds feature with id `featureID`, name `name`, and feature variation type `featureType` to the group with id `parentGroupID` in the interval $[t_n, t_m)$. `featureID` must be fresh, and the name cannot belong to any other feature in the model during the interval. The parent group must exists during the interval, and the types of the feature and the parent group must be compatible. If the feature has type MANDATORY, then the parent group must have type AND.

- **addGroup(groupID, parentFeatureID, groupType)** from t_n to t_m
  Adds group with id `groupID` and type `groupType` to the feature with id `parentFeatureID` during the interval $[t_n, t_m)$. The group ID must be fresh, and the parent feature must exist during the interval.

- **removeFeature(featureID)** at time t_n
  Removes the feature with ID `featureID` from the feature model at t_n (does not affect possible reintroductions). The FEATURES map in the original plan must contain a mapping $[$`featureID` $\mapsto$ (EXISTENCE, ...)$]$ such that $[t_i, t_j) \in$ EXISTENCE with $t_i \leq t_n \leq t_j$. The feature must not have any child groups during $[t_n, t_j)$. After removing the feature, all interval mappings should be updated from $[t_i, t_j)$ to $[t_i, t_n)$.

- **removeGroup(groupID)** at time t_n
  Removes the group with ID `groupID` from the feature model at t_n (does not affect possible reintroductions). The GROUPS map in the original plan must contain a mapping $[$`groupID` $\mapsto$ (EXISTENCE, ...)$]$ such that $[t_i, t_j) \in$ EXISTENCE with $t_i \leq t_n \leq t_j$. The group must not have any child features during $[t_n, t_j)$. After removing the group, all interval mappings should be updated from $[t_i, t_j)$ to $[t_i, t_n)$.

$$
\begin{aligned}
&(\{\ [\,\text{Washing Machine} \mapsto [\,[t_0,\infty) \mapsto 0\,]\,]\\
&\quad ,[\,\text{Washer} \mapsto [\,[t_0,\infty) \mapsto 1\,]\,]\\
&\quad ,[\,\text{Dryer} \mapsto [\,[t_5,\infty) \mapsto 2\,]\,]\ \}\\[1em]
&,\{\ [\,0 \mapsto (\{[t_0,\infty)\},\\
&\qquad \{[\,[t_0,\infty) \mapsto \text{Washing Machine}\,]\},\\
&\qquad \{[\,[t_0,\infty) \mapsto \textsc{mandatory}\,]\},\\
&\qquad \emptyset,\\
&\qquad \{[\,[t_0,\infty) \mapsto 10\,]\})\,]\\
&\quad ,[\,1 \mapsto (\{[t_0,\infty)\},\\
&\qquad \{[\,[t_0,\infty) \mapsto \text{Washer}\,]\},\\
&\qquad \{[\,[t_0,\infty) \mapsto \textsc{mandatory}\,]\},\\
&\qquad \{[\,[t_0,\infty) \mapsto 10\,]\},\\
&\qquad \emptyset)\,]\\
&\quad ,[\,2 \mapsto (\{[t_5,\infty)\},\\
&\qquad \{[\,[t_5,\infty) \mapsto \text{Dryer}\,]\},\\
&\qquad \{[\,[t_5,\infty) \mapsto \textsc{optional}\,]\},\\
&\qquad \{[\,[t_5,\infty) \mapsto 10\,]\},\\
&\qquad \emptyset)\,]\}\\[1em]
&,\ \{\ [10 \mapsto (\{[t_0,\infty)\},\\
&\qquad \{[\,[t_0,\infty) \mapsto \textsc{and}\,]\},\\
&\qquad \{[\,[t_0,\infty) \mapsto 0\,]\},\\
&\qquad \{[\,[t_0,\infty) \mapsto 1\,],\ [\,[t_5,\infty) \mapsto 2\,]\})]\\
&\quad \})
\end{aligned}
$$

Figure 2.2: Small temporal feature model

- **moveFeature(featureID, targetGroupID)** at t_n
  Moves the feature with id `featureID` to the group with ID `targetGroupID`. The move cannot be done if it introduces a cycle; that is, if the target group is in the feature's subtree. The feature's subtree is moved along with it. Parent group and child feature mappings are updated accordingly.

- **moveGroup(groupID, targetFeatureID)**
  Moves the group with id `groupID` to the feature with ID `targetFeatureID`. Very similar to **moveFeature**.

- **changeFeatureVariationType(featureID, newType)**
  Changes the feature variation type of the feature with ID `featureID` to `newType`. If the new type is MANDATORY, the parent group type must be AND.

- **changeGroupVariationType(groupID, newType)**
  Changes the group variation type of the group with ID `groupID` to `newType`. If the new type is OR or ALTERNATIVE, make sure that no child feature has type MANDATORY.

- **changeFeatureName(featureID, name)**
  Changes the name of the feature with ID `featureID` to `name`. No other feature may have the same name.

TODO: Look at possibilities for changing intervals; extending, restricting, and moving. This would be equivalent to removing/moving operations in the summer project semantics.

2.2 Define the scope

What is the scope?

Given a sound plan P and an operation associated with a timepoint O, the scope is the part of P that *may be affected* by adding O. The scope must be defined in two dimensions:

Time

Which timepoints of the plan may be affected by the change?

Space

Which parts of the feature model may be affected within the time scope?

Operation scopes

We define the *minimal* scope for each operation.

- **addFeature**(`featureID`, `parentGroupID`, `name`, `featureType`) from t_n to t_m
  TODO: Give an argument why the parent group is the spatial scope of this rule When we add a feature from t_n to t_m, it is quite obvious that the scope in time will be $[t_n, t_m)$, since this is the interval in which the feature will exist. The spatial scope must be only the parent group: If the group type changes to a conflicting one, the operation is unsound. If the parent group is removed, we have an orphaned feature, which is also illegal. But what about the name? If we only stored name information inside features, we would have to check every feature in the whole interval for this change. However, since we store information about names in a separate map, we can look up the name and check that it is not in use during the interval. TODO: Consider moving last part to data structure definitions

- **addGroup**(`groupID`, `parentFeatureID`, `groupType`) from t_n to t_m
  The scopes are very similar in this and the preceding rule. The scope in time is $[t_n, t_m)$, and the scope in space is the parent feature, for which the only conflicting event is removal – the types of a group and its parent never conflict.

- **removeFeature**(`featureID`) at t_n
  TODO: Explain why the temporal scope is the way it is If the original interval containing t_n in which the feature exists inside the feature model is $[t_m, t_k)$, then the temporal scope is $[t_n, t_k)$ - from the feature is removed until it would have been removed anyway TODO: rephrase. Since the feature is removed at t_k in the original plan, and the original plan is sound as we assume, removing the feature earlier may only affect the plan in the interval between these two time points.
  The spatial scope must be the feature's subtree. If the feature has or will have a child group during the interval, then it cannot be removed. Otherwise, there are no conflicts.

- **removeGroup**(`groupID`) at t_n
  Extremely similar to **removeFeature**.

- **moveFeature**(`featureID`, `targetGroupID`) at t_n
  If t_m is the time at which the feature is next moved in the original plan, the temporal scope is $[t_n, t_m)$, since this operation only affects the plan within this interval.
  The spatial scope is discussed in more detail in the **move feature algo**. This scope is the largest and hardest to define, because we have

to detect cycles. The scope is defined by the feature and its ancestors, as well as target group and its ancestors, which may change during the intervals. It is not necessary to look at all ancestors, only the ones which `feature` and `targetGroup` do not have in common. As usual, conflicting types and removal must be considered in addition to cycles.

- **moveGroup**(`groupID`, `targetFeatureID`) at t_n
  See moveFeature. Very similar.

- **changeFeatureVariationType**(`featureID`, `newType`) at t_n
  Temporal scope: $[t_n, t_m)$ if t_m is the next time point at which the feature's type changes or when feature is (next) removed.
  Spatial scope: The only possibly conflicting thing in the feature model is the parent group's type. At no point must the feature have type 'mandatory' and the parent group have type 'alternative' or 'or'. Thus, the spatial scope is the parent group.

- **changeGroupVariationType**(`groupID`, `newType`) at t_n Temporal scope: Same as previous.
  The spatial scope are the group's child features; the possible conflict is the same as with changeFeatureType.

- **changeFeatureName**(`featureID`, `name`) at t_n Temporal scope: Same as previous.
  Spatial scope: The name. If it already exists within the feature model during the interval, then the change is invalid.

TODO: deal with batch operations/reverting a change: It is currently impossible to *extend* an interval; If a feature exists during $[t_3, t_5)$, it is impossible to change the plan such that it exists during $[t_3, t_6)$ instead. Don't really know how to fix that, except maybe adding an operation. In Figure 2.2 on page 11, if we try to change the name of feature 1 to Dryer at t_2, intending to change it back before Dryer is added, then these semantics will reject the first change, as two features will have the name Dryer during $[t_5, \infty)$. The paradox would be righted once we add that the name will change back to Washer at t_4. There are workarounds for this, for instance changing the name of feature 2 to some temporary placeholder, making the changes to feature 1, and then changing feature 2 back. This, however, seems too cumbersome. Hopefully this use case is not common enough that most users will suffer for it, but it is definitely an example of the semantics being too strict.

2.3 SOS rules

TODO: Consider having two rules for each operation; one for validating and one for updating. Alternatively use functions on the right-hand side of $\longrightarrow$.

TODO: Remember premises = above the line

TODO: Repeat definition of syntax/motivate use of weird constructs

SOS rules TODO: explain what SOS rules are used for

The rules are on the form

(Rule-Label)

$$\frac{\text{Premises}}{S \longrightarrow S'}$$

where S is the state, and S' is the new state after the rule is applied. The rule can only be applied if all the premises hold. In this thesis, the state is always on the form **operation** $\triangleright$ $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$, where **operation** denotes the change we intend to make to the temporal feature model $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$. The new state is always on the form $(\text{NAMES}', \text{FEATURES}', \text{GROUPS}')$, where the maps have been updated according to the semantics of the operation. The premises ensure that an operation can only be applied if some conditions hold; for instance the **Add-Feature** rule 2.3 contains premises verifying that the feature does not already exist when we wish to add it.

2.3.1 Add feature rule

(Add-Feature)

$$\frac{\begin{array}{c} [t_n, t_m) \notin_{\lesseqgtr} F_e \qquad [t_n, t_m) \in_{\leq} G_e \qquad \text{NAMES}\,[\texttt{name}]\,[t_n, t_m) = \emptyset \\ \text{FEATURES}\,[\texttt{fid}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c) \\ \text{GROUPS}\,[\texttt{parentGroupID}] = (G_e,\, G_t,\, G_p,\, G_c) \\ \forall \texttt{gt}_{\in G_t[t_n, t_m)}(\texttt{compatibleTypes}(\texttt{gt}, \texttt{type})) \end{array}}{\begin{array}{c} \textbf{addFeature}(\texttt{fid}, \texttt{name}, \texttt{type}, \texttt{parentGroupID}) \text{ at } [t_n, t_m) \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\ \longrightarrow \\ (\text{NAMES}\,[\texttt{name}]\,[t_n, t_m) \leftarrow \texttt{fid}, \\ \text{FEATURES}\,[\texttt{fid}] \leftarrow \texttt{setFeatureAttributes}(\text{FEATURES}\,[\texttt{fid}], [t_n, t_m), \texttt{name}, \texttt{type}, \texttt{parentGroupID}), \\ \text{GROUPS}\,[\texttt{parentGroupID}] \leftarrow \texttt{addChildFeature}(\text{GROUPS}\,[\texttt{parentGroupID}], [t_n, t_m), \texttt{fid})) \end{array}}$$

Figure 2.3

Figure 2.3 describes the semantics of the **addFeature** operation. To add a feature during the interval $[t_n, t_m)$, its ID cannot exist exist during the

```
compatibleTypes(AND, _) = True
compatibleTypes(_, optional) = False
compatibleTypes(_, _) = True
```

Figure 2.4

```
setFeatureAttributes($(F_e, F_n, F_t, F_p, F_c)$, $[t_{start}, t_{end})$, name, type
                     , parentGroupID$)$
  = (  $F_e \cup [t_{start}, t_{end})$
     , $F_n[t_{start}, t_{end}) \leftarrow$ name
     , $F_t[t_{start}, t_{end}) \leftarrow$ type
     , $F_p[t_{start}, t_{end}) \leftarrow$ parentGroupID
     , $F_c$ )
```

Figure 2.5

interval ($[t_n, t_m) \not\in_{\lesseqgtr} F_e$). The parent feature must exist ($[t_n, t_m) \in_{\leq} G_e$), and the types it has during the interval must be compatible with the type of the added feature ($\forall \mathtt{gt}_{\in G_t[t_n,t_m)}(\mathtt{compatibleTypes}(\mathtt{gt}, \mathtt{type}))$). The name of the feature must not be in use during the interval (NAMES $[\mathtt{name}]$ $[t_n, t_m) = \emptyset$). Notice that the default value in the FEATURES map lets us treat a failed lookup as a feature, thus allowing us to express the semantics of adding a feature using only one rule.

To make the rule tidier, we use three helper functions: `compatibleTypes` (Figure 2.4), `setFeatureAttributes` (Figure 2.5), and `addChildFeature` (Figure 2.6).

2.3.2 Add group rule

The rule in figure 2.7 describes the conditions which must be in place to add a (pre-existing or fresh) group to the FMEP during an interval $[t_n, t_m)$. The group must not already exist in the plan during the interval $[t_n, t_m) \not\in_{\lesseqgtr} G_e$, and the parent feature must exist for the duration of the interval $[t_n, t_m) \in_{\leq} F_e$. The group ID is added to the parent feature's map of child groups with the interval as key, and the attributes specified are added to the group entry in the GROUPS map.

```
addChildFeature($(G_e, G_t, G_p, G_c), [t_{start}, t_{end})$, fid$)$
  = $\Big(G_e, G_t, G_p, G_c[t_{start}, t_{end}) \overset{\cup}{\leftarrow}$ fid$\Big)$
```

Figure 2.6

(Add-Group)

$$\frac{\begin{array}{c}[t_n, t_m) \not\in_{\underline{\gtrless}} G_e \qquad [t_n, t_m) \in_{\leq} F_e \\ \textsc{groups}\,[\texttt{groupID}] = (G_e,\ G_t,\ G_p,\ G_c) \\ \textsc{features}\,[\texttt{parentFeatureID}] = (F_e,\ F_n,\ F_t,\ F_p,\ F_c)\end{array}}{\begin{array}{c}\textbf{addGroup}(\texttt{groupID}, \texttt{type}, \texttt{parentFeatureID})\ \text{at}\ [t_n, t_m) \triangleright \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ (\textsc{names}, \\ \textsc{features}\,[\texttt{parentFeatureID}] \leftarrow (F_e,\ F_n,\ F_t,\ F_p,\ F_c \cup [\,[t_n, t_m) \mapsto \texttt{groupID}\,]), \\ \textsc{groups}\,[\texttt{groupID}] \leftarrow \texttt{setGroupAttributes}(\textsc{groups}\,[\texttt{groupID}]\,, \texttt{type}, \texttt{parentFeatureID}))\end{array}}$$

Figure 2.7

```
setGroupAttributes($(G_e, G_t, G_p, G_c)$, $[t_{start}, t_{end})$, type
                   , parentFeatureID)
  = ( $G_e \cup [t_{start}, t_{end})$
    , $G_t[t_{start}, t_{end})$ $\leftarrow$ type
    , $G_p[t_{start}, t_{end})$ $\leftarrow$ parentFeatureID
    , $G_c$)
```

Figure 2.8

```
addChildGroup$\left((F_e, F_n, F_t, F_p, F_c)\right.$, $[t_{start}, t_{end})$, gid$\left.\right)$
  = $\Big(F_e,\ F_n,\ F_t,\ F_p,\ F_c[t_{start}, t_{end}) \overset{\cup}{\leftarrow}$ gid$\Big)$
```

Figure 2.9

(Remove-Feature)

$$\frac{\begin{array}{c} F_e\,[t_n]_{\leq} = \{[t_{e_1}, t_{e_2})\} \qquad F_c[t_n, t_{e_2}) = \varnothing \\ F_n[t_n, t_{e_2}) = \{\texttt{name}\} \qquad F_t[t_n, t_{e_2}) = \{\texttt{type}\} \qquad F_p[t_n, t_{e_2}) = \{\texttt{parentGoupID}\} \\ \text{FEATURES}\,[\texttt{featureID}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c) \\ \text{GROUPS}\,[\texttt{parentGroupID}] = (G_e,\, G_t,\, G_p,\, G_c) \end{array}}{\begin{array}{c} \textbf{removeFeature}\,(\texttt{featureID})\ \text{at}\ t_n \rhd \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\ \longrightarrow \\ \big(\text{NAMES}\,[\texttt{name}] \leftarrow \texttt{clampInterval}(\text{NAMES}\,[\texttt{name}]\,, t_n), \\ \text{FEATURES}\,[\texttt{featureID}] \leftarrow \texttt{clampFeature}(\text{FEATURES}\,[\texttt{featureID}]\,, t_n), \\ \text{GROUPS}\,[\texttt{parentGroupID}] \leftarrow \texttt{removeFeatureAt}\,(\text{GROUPS}\,[\texttt{parentGroupID}]\,, \texttt{featureID}, t_n)\,\big) \end{array}}$$

Figure 2.10

2.3.3 Remove feature rule

Figure 2.10 shows the semantics of removing a feature with ID `featureID` at time t_n. We find the time point when the feature was to be removed in the original plan by looking up the interval containing t_n in the feature's EXISTENCE set $[t_{e_1}, t_{e_2})$. The interval in which the new plan is different from the original is then $[t_n, t_{e_2})$. We verify that the feature does not have any child groups during the affected interval ($F_c[t_n, t_{e_2}) = \varnothing$). We furthermore check that the feature has only a single name, type, and parent during the interval. This means that the original plan did not change the feature's name, type, or parent during this time. If these conditions all hold, we update the temporal feature model by clamping all the relevant intervals to t_n, i.e. shortening them to end at t_n.

2.3.4 Remove group rule

The **Remove-Group** rule in figure 2.18 describes the semantics of removing a group in a temporal feature model. The temporal scope is identified as the existence interval containing the time point for removal. In that interval, the group may not have any children, and there cannot be plans to change the type or move the group within the interval. We check the latter by looking up the type and parent feature during the interval; if the set contains only one type/parent feature then the type and parent feature do not change.

We use the `clampInterval` (figure 2.11 on the next page), `clampIntervalValue` (figure 2.12 on the following page), and `clampGroup` (figure 2.15 on the next page) helper functions to update the temporal feature model. The `clampInterval` function takes an interval map with non-overlapping keys and a time point t_c, and updates the interval key containing t_c to end at

```
clampInterval($\text{MAP}$, $t_c$)
  = let $\{[t_{start}, t_{end})\} \leftarrow \text{MAP}\,[t_c]_{\leq}$
        $\{v\} \leftarrow \text{MAP}\,[t_c]$
        $\text{MAP}' \leftarrow \text{MAP} \setminus [t_{start}, t_{end})$
    in $\text{MAP}'[t_{start}, t_c) \leftarrow v$
```

Figure 2.11

```
clampIntervalValue($\text{MAP}$, $t_c$, $v$)
  = let $\{[t_{start}, t_{end})\} \leftarrow \text{MAP}\,[t_c]^{v}_{\leq}$
        $\text{MAP}' \leftarrow \text{MAP} \setminus^{v} [t_{start}, t_{end})$
    in $\text{MAP}'[t_{start}, t_c) \overset{\cup}{\leftarrow} v$
```

Figure 2.12

```
clampSetInterval($\text{IS}$, $t_c$)
  = let $\{[t_{start}, t_{end})\} \leftarrow \text{IS}\,[t_c]_{\leq}$
        $\text{IS}' \leftarrow \text{IS} \setminus [t_{start}, t_{end})$
    in $\text{IS}' \cup \{[t_{start}, t_c)\}$
```

Figure 2.13

```
clampFeature($(F_e, F_n, F_t, F_p, F_c)$, $t_c$)
  = (clampSetInterval($F_e, t_c$)
    ,clampInterval($F_n, t_c$)
    ,clampInterval($F_t, t_c$)
    ,clampInterval($F_p, t_c$)
    ,$F_c$)
```

Figure 2.14

```
clampGroup($(G_e, G_t, G_p, G_c)$, $t_c$)
  = (clampSetInterval($G_e$)
    ,clampInterval($G_t, t_c$)
    ,clampInterval($G_p, t_c$)
    ,$G_c$)
```

Figure 2.15

```
removeFeatureAt($(G_e, G_t, G_p, G_c)$,featureID,$t_c$)
  = ($G_e, G_t, G_p$
    ,clampIntervalValue ($G_c, t_c$,featureID) )
```

Figure 2.16

```
removeGroupAt($(F_e, F_t, F_p, F_c)$,groupID,$t_c$)
  = ($F_e, F_n, F_t, F_p$
    ,clampIntervalValue ($F_c, t_c$,groupID) )
```

Figure 2.17

t_c. `clampIntervalValue` does the same, but for interval maps with overlapping keys and set values. It takes an interval map, a time point t_c, and a value v, and shortens the interval key containing t_c and v to end at t_c. `clampSetInterval` takes an interval set with non-overlapping values and a time point t_c, and shortens the interval containing t_c.

(Remove-Group)

$$\frac{\begin{array}{c} G_e\,[t_n]_{\leq} = \{[t_{e_1}, t_{e_2})\} \qquad G_c[t_n, t_{e_2}) = \emptyset \\ G_t[t_n, t_{e_2}) = \{\texttt{type}\} \qquad G_p[t_n, t_{e_2}) = \{\texttt{parentFeatureID}\} \\ \text{GROUPS}\,[\texttt{groupID}] = (G_e,\, G_t,\, G_p,\, G_c) \\ \text{FEATURES}\,[\texttt{parentFeatureID}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c) \end{array}}{\begin{array}{c} \textbf{removeGroup}\ (\texttt{groupID})\ \text{at}\ t_n \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\ \longrightarrow \\ (\text{NAMES}\,[\texttt{name}] \leftarrow \texttt{clampInterval}(\text{NAMES}\,[\texttt{name}]\,, t_n), \\ \text{FEATURES}\,[\texttt{parentFeatureID}] \leftarrow \texttt{removeGroupAt}\,(\text{FEATURES}\,[\texttt{parentFeatureID}]\,, \texttt{groupID}, t_n)\,, \\ \text{GROUPS}\,[\texttt{groupID}] \leftarrow \texttt{clampGroup}(\text{GROUPS}\,[\texttt{groupID}]\,, t_n)) \end{array}}$$

Figure 2.18

2.3.5 Algorithm for detecting cycles resulting from move operations

Compared with the other operations (e.g. add feature, remove feature, etc.), move feature requires extensive verification. Following is a description of an algorithm intended to ensure that adding a **moveFeature** or **moveGroup** operation results in a sound plan. For simplicity, we abstract away from groups and features and view the combination of the two as nodes.

Let n be the node to be moved and c_1 the target node, i.e. n's new parent node. Furthermore, let t_1 be the time point at which this operation is inserted, and t_e the time point where n is moved next or removed, or ∞. We use the function $\texttt{ancestors}(TFM, \texttt{node}, \texttt{time})$ which takes the temporal feature model, a node and a time point and returns a list of `node`'s ancestors at time point `time`.

First, check whether $n \in \texttt{ancestors}(TFM, c_1, t_1)$. If this is the case, report that the move causes a cycle and terminate.

Next, find a list of critical nodes. Let $A_n = \texttt{ancestors}(TFM, n, t_1) = [a_1, a_2, \ldots, SN, \ldots, r]$ and $A_{c_1} = \texttt{ancestors}(TFM, c_1, t_1) = [c_2, c_3, \ldots, c_n, SN, \ldots, r]$ with SN the first common ancestor of n and c_1. The list of critical nodes is then $C = [c_1, c_2, \ldots, c_n]$, which is essentially the list of n's new ancestors after the move.

Repeat this step until the algorithm terminates:

Look for the first move of one of the critical nodes. If no such moves occur until t_e, the operation causes no paradoxes, and the algorithm terminates successfully. Suppose there is a 'move' operation scheduled for t_k, with $t_1 < t_k < t_e$, where c_i is moved to k. There are two possibilities:

1. k is in n's subtree, which is equivalent to $n \in \texttt{ancestors}(TFM, k, t_k)$. Report that the move caused a cycle and terminate. 2. k is not in n's subtree, so this move is safe. Let $A_k = \texttt{ancestors}(TFM, k, t_k) = [k_1, k_2, \ldots, k_n, SN', \ldots, r]$, with SN′ the first common element of A_k and A_n. Update the list of critical nodes to $[c_1, \ldots, c_i, k_1, \ldots, k_n]$.

```
ancestors((NAMES, FEATURES, GROUPS), featureID, $t_n$)
  = let  $(F_e, F_n, F_t, F_p, F_c) \leftarrow$ FEATURES [featureID]
         parentGroup $\leftarrow F_p\,[t_n]$
     in
      case parentGroup of
        { parentGroupID } $\rightarrow$
          parentGroupID : ancestors((NAMES, FEATURES, GROUPS), parentGroupID, $t_n$)
        $\emptyset \rightarrow$ []
ancestors((NAMES, FEATURES, GROUPS), groupID, $t_n$)
  = let  $(G_e, G_t, G_p, G_c) \leftarrow$ GROUPS [groupID]
         { parentFeatureID } $\leftarrow G_p\,[t_n]$
     in
        parentFeatureID : ancestors((NAMES, FEATURES, $groups$), parentFeatureID, $t_n$)
```

Figure 2.19

2.3.6 Move feature rule

See figure 2.20 on the following page for the semantics of the **moveFeature** operation. The premise ¬`createsCycle` refers to the algorithm described in subsection 2.3.5 on the previous page. The algorithm is not described as a function here, as it is easier to understand written in natural language. An implementation of it can be found in the TODO: appendix.

2.3.7 Move group rule

See figure 2.21 on page 23 for the semantics of the **moveGroup** operation. The semantics is similar to the one for the **moveFeature** operation, but it differs in that it does not have a check for types. This is because there can only be a conflict between a parent group and a child feature, not a parent feature and a child group. Since only the latter relation changes in this rule, it is not necessary to check that the types are compatible.

TODO: This is hard

(Move-Feature)

$$\frac{\begin{array}{c}\neg\texttt{createsCycle} \qquad F_p\,[t_n]_{\leq} = \{[t_{p_1}, t_{p_2})\} \qquad F_p[t_n, t_{p_2}) = \{\texttt{oldParentID}\} \\ \forall \texttt{gt}_{\in G_t[t_n,t_m)}(\texttt{compatibleTypes}(\texttt{gt},\texttt{type})) \\ \text{FEATURES}\,[\texttt{featureID}] = (F_e,\ F_n,\ F_t,\ F_p,\ F_c) \\ \text{GROUPS}\,[\texttt{newParentID}] = (G_e,\ G_t,\ G_p,\ G_c)\end{array}}{\begin{array}{c}\textbf{moveFeature}\,(\texttt{featureID},\ \texttt{newParentID})\text{ at } t_n \rhd \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\ \longrightarrow \\ (\text{NAMES}, \\ \text{FEATURES}\,[\texttt{featureID}] \leftarrow (F_e,\ F_n,\ F_t,\ \texttt{clampInterval}(F_p, t_n)[t_n, t_{p_2}) \leftarrow \texttt{newParentID},\ F_c)\,, \\ (\text{GROUPS}\,[\texttt{newParentID}] \\ \leftarrow \texttt{addChildFeature}(\text{GROUPS}\,[\texttt{newParentID}]\,, [t_n, t_{p_2}), \texttt{featureID}))\,[\texttt{oldParentID}] \\ \leftarrow \texttt{removeFeatureAt}\,(\text{GROUPS}\,[\texttt{oldParentID}]\,, \texttt{featureID}, t_n)\end{array}}$$

Figure 2.20

2.3.8 Change feature variation type rule

The rule in figure 2.22 on the next page shows the semantics of changing the feature variation type of the feature with ID `featureID` at time t_n. The first expression above the line ($F_t\,[t_n]_{\leq} = \{[t_{t_1}, t_{t_2})\}$) identifies the upper bound of the temporal scope, t_{t_2}. This is when the feature type was originally planned to change. The next line may be hard to read, but its intent is easier to understand. It checks that all the types a parent group has *while it is the parent of the feature* has a type which is compatible with the new type of the feature. If everything above the line is true, then the FEATURES map is updated at `featureID` by shortening the interval key for the original type at t_n, and assigning the new type to the affected interval $[t_n, t_{t_2})$.

TODO: make it more readable

2.3.9 Change group variation type rule

The rule in figure 2.24 on page 24 is similar to the **changeFeatureVariationType** rule in figure 2.22 on the next page, and shows the semantics of changing the type of a group. In a similar way to the aforementioned **changeFeatureVariationType** rule, it verifies that the types of all the child groups during the affected interval are compatible with the new group type.

(Move-Group)

$$\dfrac{\begin{array}{c} \neg\texttt{createsCycle} \qquad G_p\,[t_n]_{\leq} = \big\{[t_{p_1}, t_{p_2})\big\} \qquad G_p[t_n, t_{p_2}) = \{\texttt{oldParentID}\} \\ \textsc{groups}\,[\texttt{groupID}] = \big(G_e,\, G_t,\, G_p,\, G_c\big) \\ \textsc{features}\,[\texttt{newParentID}] = \big(F_e,\, F_n,\, F_t,\, F_p,\, F_c\big) \end{array}}{\begin{array}{c} \textbf{moveGroup}\,(\texttt{groupID, newParentID})\ \text{at}\ t_n \rhd \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ \big(\textsc{names}, \\ \big(\textsc{features}\,[\texttt{newParentID}] \\ \leftarrow \texttt{addChildGroup}(\textsc{features}\,[\texttt{newParentID}]\,, [t_n, t_{p_2}), \texttt{groupID})\big)\ [\texttt{oldParentID}] \\ \leftarrow \texttt{removeGroupAt}\,(\textsc{features}\,[\texttt{oldParentID}]\,, \texttt{groupID}, t_n)\,, \\ \textsc{groups}\,[\texttt{groupID}] \leftarrow \big(G_e,\, G_n,\, G_t,\, \texttt{clampInterval}(G_p, t_n)[t_n, t_{p_2}) \leftarrow \texttt{newParentID},\, G_c\big) \end{array}}$$

Figure 2.21

(Change-Feature-Variation-Type)

$$\dfrac{\begin{array}{c} \texttt{featureID} \neq \texttt{rootID} \qquad F_t\,[t_n]_{\leq} = \{[t_{t_1}, t_{t_2})\} \\ \forall [t_{p_1}, t_{p_2}) \in F_p[t_n, t_{t_2})_{\lessgtr} \forall p \in F_p[t_{p_1}, t_{p_2}) \forall t \in \texttt{getTypes}\,\Big(\textsc{groups}\,[p]\,, \big\langle [t_{p_1}, t_{p_2}) \big\rangle_{t_n}^{t_{t_2}}\Big) \cdot \big(\texttt{compatibleTypes}(t, \texttt{type} \\ \textsc{features}\,[\texttt{featureID}] = \big(F_e,\, F_n,\, F_t,\, F_p,\, F_c\big) \end{array}}{\begin{array}{c} \textbf{changeFeatureVariationType}\,(\texttt{featureID}, \texttt{type})\ \text{at}\ t_n \rhd \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ (\textsc{names}, \\ \textsc{features}\,[\texttt{featureID}] \leftarrow \big(F_e,\, F_n,\, \texttt{clampInterval}(F_t, t_n)[t_n, t_{t_2}) \leftarrow \texttt{type},\, F_p,\, F_c\big)\,, \\ \textsc{groups}) \end{array}}$$

Figure 2.22

2.3.10 Change feature name

The semantics of changing the name of a feature are shown in the **Change-Feature-Name** rule in figure 2.25 on the following page. The old name and the next planned name change are identified on the first line ($F_n\,[t_n] = \{\texttt{oldName}\}$ and $F_n\,[t_n]_{\leq} = \{[t_{n_1}, t_{n_2})\}$ respectively). Since the name must not be in use during the temporal scope, we verify that looking up the new name in the NAMES map returns an empty set. The NAMES map is updated by shortening the interval for the old name to end at t_n, and assigning the feature ID to the new name during the temporal scope. Furthermore, the FEATURES map is updated at the feature ID, shortening the interval for the old name and assigning the new name to the temporal scope.

$$
\begin{array}{l}
\texttt{getTypes}\big((G_e, G_t, G_p, G_c),\ [t_n, t_m)\big) = G_t[t_n, t_m) \\
\texttt{getTypes}\big((F_e, F_n, F_t, F_p, F_c),\ [t_n, t_m)\big) = G_t[t_n, t_m)
\end{array}
$$

Figure 2.23

(Change-Group-Variation-Type)

$$
\begin{array}{c}
G_t\ [t_n]_{\leq} = \{[t_{t_1}, t_{t_2})\} \\[1em]
\begin{array}{l}
\forall [t_{c_1}, t_{c_2}) \in G_c[t_n, t_{t_2})_{\underline{\lessgtr}} \\
\forall c \in \bigcup G_c[t_{c_1}, t_{c_2}) \\
\forall t \in\ \texttt{getTypes}\left(\text{FEATURES}\,[c]\,, \langle [t_{c_1}, t_{c_2}) \rangle_{t_n}^{t_{t_2}}\right) \\
\quad \big(\texttt{compatibleTypes}(\texttt{type}, t)\big)
\end{array} \\[1em]
\text{GROUPS}\,[\texttt{groupID}] = \big(G_e,\ G_t,\ G_p,\ G_c\big) \\
\hline
\textbf{changeGroupVariationType}\,(\texttt{groupID}, \texttt{type})\ \text{ at } t_n \triangleright \\
(\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\
\longrightarrow \\
(\text{NAMES}, \text{FEATURES}, \\
\text{GROUPS}\,[\texttt{groupID}] \leftarrow \big(G_e,\ \texttt{clampInterval}(G_t, t_n)[t_n, t_{t_2}) \leftarrow \texttt{type},\ G_p,\ G_c\big))
\end{array}
$$

Figure 2.24

(Change-Feature-Name)

$$
\begin{array}{c}
F_n\ [t_n] = \{\texttt{oldName}\} \qquad F_n\ [t_n]_{\leq} = \{[t_{n_1}, t_{n_2})\} \\
\text{NAMES}\,[\texttt{name}]\ [t_n, t_{n_2}) = \emptyset \\
\text{FEATURES}\,[\texttt{featureID}] = \big(F_e,\ F_n,\ F_t,\ F_p,\ F_c\big) \\
\hline
\textbf{changeFeatureName}\,(\texttt{featureID}, \texttt{name})\ \text{ at } t_n \triangleright \\
(\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\
\longrightarrow \\
((\text{NAMES}\,[\texttt{name}]\ [t_n, t_{n_2}) \leftarrow \texttt{featureID})\ [\texttt{oldName}] \leftarrow \texttt{clampInterval}(\text{NAMES}\,[\texttt{oldName}]\,, t_n), \\
\text{FEATURES}\,[\texttt{featureID}] \leftarrow \big(F_e,\ \texttt{clampInterval}(F_n, t_n)[t_n, t_{n_2}) \leftarrow \texttt{name},\ F_t,\ F_p,\ F_c\big)\,, \\
\text{GROUPS})
\end{array}
$$

Figure 2.25

Chapter 3

Soundness

TODO: and completeness? The operations in themselves are not complete, in the sense that there is not a one-to-one correspondence between the operations defined on temporal feature models (temporal operations) and the operations defined on feature models (model operations). However, the temporal operations are more restricted than the model operations, and any valid change within that subset to an evolution plan would be accepted by the rules. Should I try to prove completeness as well?

TODO: Root feature requirements

3.1 Soundness for temporal feature models

The temporal feature model can be viewed as a sequence of feature models associated with time points. A feature model has strict structural requirements, and the definition of a paradox is a feature model that violates these requirements. In this context, soundness means that if a rule accepts a modification, realising the modified plan results in a sequence of feature models where each is structurally sound. The soundness analysis in this chapter assumes that the original plan is sound; i.e. all resulting feature models fulfil the structural requirements.

According to Motzfeldt et al. 2020, the structural requirements (well-formedness rules) are

WF1 A feature model has exactly one root feature.

WF2 The root feature must be mandatory.

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

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

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

WF6 Each group must have exactly one parent feature.

WF7 Groups with types ALTERNATIVE or OR must not contain MANDATORY features.

WF8 Groups with types ALTERNATIVE or OR must contain at least two child features.

One of these requirements is not upheld in this thesis, namely ***WF8***. This is because it would cause unnecessary clutter in the rules, although it could be verified quite easily in an actual implementation.

TODO: This may not be a good idea, since the reasoning for scope blends with the reasoning for the other checks. Maybe divide into validate and apply, like in the implementation? The proof for soundness includes two parts:

(i) proving that the temporal and spatial scopes are correct, and

(ii) that the checks within those scopes are sufficient for soundness.

For each rule, the proofs are divided into these two parts.

3.1.1 Soundness for the Add feature rule

See figure 2.3 on page 15 for the **Add-Feature** rule. Let

$$\textbf{addFeature}(\texttt{fid}, \texttt{name}, \texttt{type}, \texttt{parentGroupID}) \text{ at } [t_n, t_m) \triangleright (\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$

be the initial state. Recall that this operation adds the feature with ID `fid` to the temporal feature model $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ from t_n to t_n.

We argue that the temporal scope is $[t_n, t_m)$, since this is the only interval in which the temporal feature model is affected by the change. In other words, if we look at the temporal feature model as a sequence of feature models, the only models that may break as a result of this modifications, are the ones associated with time points between t_n and (but not including) t_m.

The well-formedness rules must hold for each feature model in the sequence; that is, for any time point t_k within the scope, the feature model at t_k must uphold all the rules.

Since we give a parent group in the operation, the new feature cannot be a root feature, so ***WF1*** is not violated. It is also obvious that the root feature does not change its type in this rule, so ***WF2*** is upheld.

However, ***WF3*** must be looked at more closely for this rule. Since we check that no other feature has the name `name` during the temporal scope, the name is unique for each affected time point. The variation type is given in the operation, and a newly added feature does not have child groups. Thus, ***WF3*** is upheld. Since the rule does not change the type of a group, ***WF4*** is not affected. As for ***WF5***, since the ID of the parent group (`parentGroupID`) is also specified in the operation, the feature is part of exactly one group. TODO: WF6!!!

Chapter 4

Application

Say something about implementation without showing the code, maybe giving pseudocode. Talk about distance between formalization and implementation. Describe examples, error messages, practical applications, how it can be used, how it detects paradoxes, how warnings can be given to users.

Chapter 5

Implementation

TODO: Move to appendix

Part III

Conclusion

How I have addressed the questions, and how I have *not* addressed the questions. Based on the assumption etc. Pinpoint other work that can be done to address the questions I don't tackle. Another master thesis can focus on presenting the input and output to the user

Bibliography

Motzfeldt, Ida Sandberg et al. (2020). “Consistency-preserving evolution planning on feature models”. In: *SPLC ’20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*. Ed. by Roberto Erick Lopez-Herrejon. ACM, 8:1–8:12. DOI: `10.1145/3382025.3414964`. URL: `https://doi.org/10.1145/3382025.3414964`.