Master Thesis

Methods for modular soundness checking 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 checking of feature model evolution plans

Ida Sandberg Motzfeldt

Master Thesis

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

Printed: Reprosentralen, University of Oslo

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

Definition 2.1 (Temporal feature model)**.** A *temporal feature model* is defined as a triple $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ where NAMES is a *map* from *names* to *feature IDs*, FEATURES is a map from feature IDs to *feature entries*, and GROUPS is a map from *group IDs* to *group entries*.

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.

2.1.2 Maps

Definition 2.2 (Map)**.** A *map* is a set of entries on the form $[\, k \mapsto v \,]$, where each key k uniquely defines a value v.

Following is the syntax for looking up a value at the key k in map MAP:

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

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 $O(1)$ TODO: $\leftarrow$ maybe not? 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.

2.1.3 Intervals

To define the map values used in temporal feature models, we must first define an *interval*.

Definition 2.3 (Interval)**.** An interval is a set of time points between a lower and an upper bound. We denote the interval using the familiar mathematical notation $[t_{\text{start}}, t_{\text{end}})$, where t_{start} is the lower bound, and

t_{end} is the upper bound. These intervals are left-closed and right-open, meaning that t_{start} is contained in the interval, and all time points until but not including t_{end}.

We say that an interval $[t_{\text{start}}, t_{\text{end}})$ *contains* the time point t_k if $t_{\text{start}} \leq t_k < t_{\text{end}}$. Two intervals $[t_n, t_m)$ and $[t_i, t_j)$ *overlap* if 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.

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 $[\mathbf{max}(t_{start}, t_l), \mathbf{min}(t_{end}, t_r))$.

2.1.4 Interval maps

Definition 2.4 (Interval map)**.** An *interval map* is a map where the key is an interval (definition 2.3).

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 will return $\{v\}$ (assuming that $t_1 < t_2 < \ldots < t_5$):

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

Figure 2.1: Interval map example

$\text{IM}\,[t_n]_{\leq}$ returns the set of keys containing time point t_n. For interval maps with non-overlapping keys, the resulting set will contain at most one element. For interval maps with set values, we define an additional function $\text{IM}\,[t_n]_{\leq}^{v}$ 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 $\text{IM}\,[[t_n, t_m)]_{\lessapprox}$ which returns all the interval keys in the map IM overlapping the interval $[t_n, t_m)$.

2.1.5 Interval sets

Definition 2.5 (Interval set)**.** An *interval set* is a set of intervals (definition 2.3).

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 $\in_{\lessapprox}$ such that $[t_n, t_m) \in_{\lessapprox}$ IS iff there exists some interval $[t_i, t_j) \in$ IS with $[t_n, t_m)$ overlapping $[t_i, t_j)$.

Notice that $\in \subseteq \in_{\leq} \subseteq \in_{\lessapprox}$, which means that if $[t_n, t_m) \in$ IS then also $[t_n, t_m) \in_{\leq}$ IS, and $[t_n, t_m) \in_{\lessapprox}$ IS. TODO: double check with respect to the next paragraph

We also define these predicates for time points t_n, so that $t_n \in_{\leq}$ IS if some interval $[t_i, t_j) \in$ IS with $t_i \leq t_n < t_j$.

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

2.1.6 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.

2.1.7 Mapping features

The FEATURES map has entries of the form [`featureID` $\mapsto$ *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.

2.1.8 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 $(\emptyset, \emptyset, \emptyset, \emptyset)$.

2.1.9 Example evolution plan

A small example of a temporal feature model can be found in Figure 2.2 on the next page. 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.

2.1.10 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

$$
\begin{aligned}
&(\{\ [\text{Washing Machine} \mapsto [\,[t_0, \infty) \mapsto 0\,]\,] \\
&\quad , [\text{Washer} \mapsto [\,[t_0, \infty) \mapsto 1\,]\,] \\
&\quad , [\text{Dryer} \mapsto [\,[t_5, \infty) \mapsto 2\,]\,]\ \} \\
&\\
&\ , \{\ [0 \mapsto (\{[t_0, \infty)\}, \\
&\qquad \{[\,[t_0, \infty) \mapsto \text{Washing Machine}\,]\}, \\
&\qquad \{[\,[t_0, \infty) \mapsto \text{MANDATORY}\,]\}, \\
&\qquad \emptyset, \\
&\qquad \{[\,[t_0, \infty) \mapsto 10\,]\})\,] \\
&\quad , [1 \mapsto (\{[t_0, \infty)\}, \\
&\qquad \{[\,[t_0, \infty) \mapsto \text{Washer}\,]\}, \\
&\qquad \{[\,[t_0, \infty) \mapsto \text{MANDATORY}\,]\}, \\
&\qquad \{[\,[t_0, \infty) \mapsto 10\,]\}, \\
&\qquad \emptyset)\,] \\
&\quad , [2 \mapsto (\{[t_5, \infty)\}, \\
&\qquad \{[\,[t_5, \infty) \mapsto \text{Dryer}\,]\}, \\
&\qquad \{[\,[t_5, \infty) \mapsto \text{OPTIONAL}\,]\}, \\
&\qquad \{[\,[t_5, \infty) \mapsto 10\,]\}, \\
&\qquad \emptyset)\,]\} \\
&\\
&\ , \{\ [10 \mapsto (\{[t_0, \infty)\}, \\
&\qquad \{[\,[t_0, \infty) \mapsto \text{AND}\,]\}, \\
&\qquad \{[\,[t_0, \infty) \mapsto 0\,]\}, \\
&\qquad \{[\,[t_0, \infty) \mapsto 1\,],\ [\,[t_5, \infty) \mapsto 2\,]\})] \\
&\quad \})
\end{aligned}
$$

Figure 2.2: Small temporal feature model

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 $[\texttt{featureID} \mapsto (\text{EXISTENCE}, \ldots)]$ 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 $[\texttt{groupID} \mapsto (\text{EXISTENCE}, \ldots)]$ 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 $[t_i, t_n)$.

- **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 time point 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 time points of the plan may be affected by the change?

Space

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

Operation scopes

We define the *minimal* scope for each operation.

- **addFeature**(`featureID`, `parentGroupID`, `name`, `featureType`) from t_n to t_m
  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 modification, are the ones associated with time points between t_n and (but not including) t_m. The spatial scope must be only the feature itself, the parent group and the name. If the group type of the parent changes to a conflicting one, the operation is unsound. If the parent group is removed, we have an orphaned feature, which is also illegal. The name is unique, so we must also verify that no other feature is using the name during the temporal scope.

- **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 group with id `groupID` and the parent feature with ID`parentFeatureID`, 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 itself, its parent group, and its subtree. If the feature has or will have a child group during the interval, then it cannot be removed. Otherwise, there are no conflicts. When modifying the temporal feature model, the feature must be removed from the parent's set of subfeatures, which is why the parent group is included in the spatial scope.

- **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** $\triangleright$ (NAMES, FEATURES, GROUPS), where **operation** denotes the change we intend to make to the temporal feature model (NAMES, FEATURES, GROUPS). The new state is always on the form (NAMES′, FEATURES′, 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)

$$\frac{\begin{array}{c}[t_n, t_m) \not\in_{\lessgtr} F_e \qquad [t_n, t_m) \in_{\leq} G_e \qquad \text{NAMES}\,[\texttt{name}]\,[[t_n, t_m)] = \emptyset \\ \text{FEATURES}\,[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c) \\ \text{GROUPS}\,[\texttt{parentGroupID}] = (G_e, G_t, G_p, G_c) \\ \forall \texttt{gt}_{\in G_t[[t_n,t_m)]}(\texttt{compatibleTypes}(\texttt{gt}, \texttt{type}))\end{array}}{\begin{array}{c}\textbf{addFeature}(\texttt{featureID}, \texttt{name}, \texttt{type}, \texttt{parentGroupID}) \text{ at } [t_n, t_m) \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\ \longrightarrow \\ (\text{NAMES}\,[\texttt{name}]\,[[t_n, t_m)] \leftarrow \texttt{featureID}, \\ \text{FEATURES}\,[\texttt{featureID}] \leftarrow \texttt{setFeatureAttributes}(\text{FEATURES}\,[\texttt{featureID}], [t_n, t_m), \\ \texttt{name}, \texttt{type}, \texttt{parentGroupID}), \\ \text{GROUPS}\,[\texttt{parentGroupID}] \leftarrow \texttt{addChildFeature}(\text{GROUPS}\,[\texttt{parentGroupID}], [t_n, t_m), \texttt{featureID}))\end{array}}$$

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

$$\begin{aligned}&\texttt{setFeatureAttributes}\big((F_e, F_n, F_t, F_p, F_c),\ [t_{start}, t_{end}),\ \texttt{name},\ \texttt{type},\ \texttt{parentGroupID}\big)\\ &\quad = (\ F_e \cup [t_{start}, t_{end})\\ &\qquad ,\ F_n\,[[t_{start}, t_{end})] \leftarrow \texttt{name}\\ &\qquad ,\ F_t\,[[t_{start}, t_{end})] \leftarrow \texttt{type}\\ &\qquad ,\ F_p\,[[t_{start}, t_{end})] \leftarrow \texttt{parentGroupID}\\ &\qquad ,\ F_c\ )\end{aligned}$$

Figure 2.5

interval ($[t_n, t_m) \not\in_{\gtrless} F_e$). 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 ($\forall \texttt{gt}_{\in G_t[[t_n,t_m)]}(\texttt{compatibleTypes}(\texttt{gt}, \texttt{type}))$). The name of the feature must not be in use during the interval ($\text{NAMES}\,[\texttt{name}]\,[[t_n, t_m)] = \emptyset$). 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 ($[t_n, t_m) \not\in_{\gtrless} G_e$), 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.

$$\begin{aligned}&\texttt{addChildFeature}\big((G_e, G_t, G_p, G_c), [t_{start}, t_{end}), \texttt{fid}\big)\\ &\quad = \Big(G_e, G_t, G_p, G_c\,[[t_{start}, t_{end})] \overset{\cup}{\leftarrow} \texttt{fid}\Big)\end{aligned}$$

Figure 2.6

(ADD-GROUP)

$$\frac{[t_n, t_m) \not\in_{\gtrless} G_e \qquad [t_n, t_m) \in_{\leq} F_e \qquad \text{GROUPS}\,[\texttt{groupID}] = (G_e, G_t, G_p, G_c) \qquad \text{FEATURES}\,[\texttt{parentFeatureID}] = (F_e, F_n, F_t, F_p, F_c)}{\begin{array}{c}\textbf{addGroup}(\texttt{groupID}, \texttt{type}, \texttt{parentFeatureID}) \text{ at } [t_n, t_m) \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\ \longrightarrow \\ (\text{NAMES}, \\ \text{FEATURES}\,[\texttt{parentFeatureID}] \leftarrow \texttt{addChildGroup}\,(\text{FEATURES}\,[\texttt{parentFeatureID}]\,, [t_n, t_m), \texttt{groupID})\,, \\ \text{GROUPS}\,[\texttt{groupID}] \leftarrow \texttt{setGroupAttributes}(\text{GROUPS}\,[\texttt{groupID}]\,, \texttt{type}, \texttt{parentFeatureID}))\end{array}}$$

Figure 2.7

$$\begin{aligned}&\texttt{setGroupAttributes}\big(\,(G_e, G_t, G_p, G_c)\,, [t_{start}, t_{end}), \texttt{type} \\ &\qquad\qquad\qquad\qquad \texttt{,parentFeatureID}\big) \\ &\quad = (\;\; G_e \cup [t_{start}, t_{end}) \\ &\qquad , \;\; G_t\,[[t_{start}, t_{end})] \;\leftarrow\; \texttt{type} \\ &\qquad , \;\; G_p\,[[t_{start}, t_{end})] \;\leftarrow\; \texttt{parentFeatureID} \\ &\qquad , \;\; G_c)\end{aligned}$$

Figure 2.8

$$\begin{aligned}&\texttt{addChildGroup}\big((F_e, F_n, F_t, F_p, F_c)\,, [t_{start}, t_{end}), \texttt{groupID}\big) \\ &\quad = \Big(F_e, F_n, F_t, F_p, F_c\,[[t_{start}, t_{end})] \overset{\cup}{\leftarrow} \texttt{groupID}\Big)\end{aligned}$$

Figure 2.9

(REMOVE-FEATURE)

$$F_e\,[t_n]_{\leq} = \{[t_{e_1}, t_{e_2})\} \qquad F_c\,[[t_n, t_{e_2})] = \emptyset$$
$$F_n\,[[t_n, t_{e_2})] = \{\texttt{name}\} \qquad F_t\,[[t_n, t_{e_2})] = \{\texttt{type}\} \qquad F_p\,[[t_n, t_{e_2})] = \{\texttt{parentGroupID}\}$$
$$\text{FEATURES}\,[\texttt{featureID}] = (F_e,\ F_n,\ F_t,\ F_p,\ F_c)$$
$$\text{GROUPS}\,[\texttt{parentGroupID}] = (G_e,\ G_t,\ G_p,\ G_c)$$

$$\textbf{removeFeature}\,(\texttt{featureID})\ \text{at}\ t_n \triangleright$$
$$(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$
$$\longrightarrow$$
$$(\text{NAMES}\,[\texttt{name}] \leftarrow \texttt{clampInterval}(\text{NAMES}\,[\texttt{name}]\,, t_n),$$
$$\text{FEATURES}\,[\texttt{featureID}] \leftarrow \texttt{clampFeature}(\text{FEATURES}\,[\texttt{featureID}]\,, t_n),$$
$$\text{GROUPS}\,[\texttt{parentGroupID}] \leftarrow \texttt{removeFeatureAt}\,(\text{GROUPS}\,[\texttt{parentGroupID}]\,, \texttt{featureID}, t_n)\,)$$

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 ($F_c\,[[t_n, t_{e_2})] = \emptyset$). 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 on page 21 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 following page), `clampIntervalValue` (figure 2.12 on the next page), and `clampGroup` (figure 2.15 on the following page) helper functions to update the temporal feature model. The

$$
\begin{aligned}
&\texttt{clampInterval}(\text{MAP}, t_c)\\
&\quad = \texttt{let } \{[t_{start}, t_{end})\} \leftarrow \text{MAP}\,[t_c]_{\leq}\\
&\qquad\quad \{v\} \leftarrow \text{MAP}\,[t_c]\\
&\qquad\quad \text{MAP}' \leftarrow \text{MAP} \setminus [t_{start}, t_{end})\\
&\qquad \texttt{in } \text{MAP}'\,[[t_{start}, t_c)] \leftarrow v
\end{aligned}
$$

Figure 2.11

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

Figure 2.12

$$
\begin{aligned}
&\texttt{clampSetInterval}(\text{IS}, t_c)\\
&\quad = \texttt{let } \{[t_{start}, t_{end})\} \leftarrow \text{IS}\,[t_c]_{\leq}\\
&\qquad\quad \text{IS}' \leftarrow \text{IS} \setminus [t_{start}, t_{end})\\
&\qquad \texttt{in } \text{IS}' \cup \{[t_{start}, t_c)\}
\end{aligned}
$$

Figure 2.13

$$
\begin{aligned}
&\texttt{clampFeature}\big((F_e, F_n, F_t, F_p, F_c)\,, t_c\big)\\
&\quad = (\texttt{clampSetInterval}(F_e, t_c)\\
&\qquad ,\texttt{clampInterval}(F_n, t_c)\\
&\qquad ,\texttt{clampInterval}(F_t, t_c)\\
&\qquad ,\texttt{clampInterval}(F_p, t_c)\\
&\qquad ,F_c)
\end{aligned}
$$

Figure 2.14

$$
\begin{aligned}
&\texttt{clampGroup}\big((G_e, G_t, G_p, G_c)\,, t_c\big)\\
&\quad = (\texttt{clampSetInterval}(G_e)\\
&\qquad ,\texttt{clampInterval}(G_t, t_c)\\
&\qquad ,\texttt{clampInterval}(G_p, t_c)\\
&\qquad ,G_c)
\end{aligned}
$$

Figure 2.15

$$
\begin{aligned}
&\texttt{removeFeatureAt}\big(\,(G_e, G_t, G_p, G_c)\,, \texttt{featureID}, t_c\big)\\
&\quad = \big(G_e, G_t, G_p\\
&\qquad ,\texttt{clampIntervalValue}\,(G_c, t_c, \texttt{featureID})\,\big)
\end{aligned}
$$

Figure 2.16

$$
\begin{aligned}
&\texttt{removeGroupAt}\big((F_e, F_n, F_t, F_p, F_c)\,, \texttt{groupID}, t_c\big)\\
&\quad = \big(F_e, F_n, F_t, F_p\\
&\qquad ,\texttt{clampIntervalValue}\,(F_c, t_c, \texttt{groupID})\,\big)
\end{aligned}
$$

Figure 2.17

`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 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)

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

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) ← FEATURES [featureID]
        parentGroup ← F_p [t_n]
    in
     case parentGroup of
       { parentGroupID } →
         parentGroupID : ancestors((NAMES, FEATURES, GROUPS), parentGroupID, t_n)
       ∅ → []
ancestors((NAMES, FEATURES, GROUPS), groupID, t_n)
  = let (G_e, G_t, G_p, G_c) ← GROUPS [groupID]
        { parentFeatureID } ← 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 next page for the semantics of the **moveFeature** operation. The premise ¬`createsCycle` refers to the algorithm described in subsection 2.3.5 on the preceding 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.

(MOVE-FEATURE)

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

Figure 2.20

2.3.7 Move group rule

See figure 2.21 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.

(MOVE-GROUP)

$$\frac{\begin{array}{c}\neg\texttt{createsCycle} \qquad G_p\,[t_n]_{\leq} = \{[t_{p_1}, t_{p_2})\} \qquad G_p\,[[t_n, t_{p_2})] = \{\texttt{oldParentID}\} \\ \text{GROUPS}\,[\texttt{groupID}] = (G_e,\, G_t,\, G_p,\, G_c) \\ \text{FEATURES}\,[\texttt{newParentID}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c)\end{array}}{\begin{array}{c}\textbf{moveGroup}\,(\texttt{groupID},\ \texttt{newParentID})\ \text{at}\ t_n \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\ \longrightarrow \\ (\text{NAMES}, \\ (\text{FEATURES}\,[\texttt{newParentID}] \\ \leftarrow \texttt{addChildGroup}(\text{FEATURES}\,[\texttt{newParentID}]\,, [t_n, t_{p_2}), \texttt{groupID}))\,[\texttt{oldParentID}] \\ \leftarrow \texttt{removeGroupAt}\,(\text{FEATURES}\,[\texttt{oldParentID}]\,, \texttt{groupID}, t_n)\,, \\ \text{GROUPS}\,[\texttt{groupID}] \leftarrow (G_e,\, G_n,\, G_t,\, \texttt{clampInterval}(G_p, t_n)\,[[t_n, t_{p_2})] \leftarrow \texttt{newParentID},\, G_c)\end{array}}$$

Figure 2.21

TODO: This is hard

(CHANGE-FEATURE-VARIATION-TYPE)

$$\frac{\begin{array}{c}\texttt{featureID} \neq \texttt{rootID} \qquad F_t\left[t_n\right]_{\leq} = \{[t_{t_1}, t_{t_2})\} \\ \forall [t_{p_1}, t_{p_2}) \in F_p\left[[t_n, t_{t_2})\right]_{\lessapprox} \\ \forall p \in F_p\left[[t_{p_1}, t_{p_2})\right] \\ \forall t \in \texttt{getTypes}\left(\text{GROUPS}\left[p\right], \left\langle [t_{p_1}, t_{p_2})\right\rangle_{t_n}^{t_{t_2}}\right) \\ \left(\texttt{compatibleTypes}(t, \texttt{type})\right) \\ \\ \text{FEATURES}\left[\texttt{featureID}\right] = \left(F_e,\, F_n,\, F_t,\, F_p,\, F_c\right)\end{array}}{\begin{array}{c}\textbf{changeFeatureVariationType}\left(\texttt{featureID}, \texttt{type}\right) \text{ at } t_n \triangleright \\ \left(\text{NAMES}, \text{FEATURES}, \text{GROUPS}\right) \\ \longrightarrow \\ \left(\text{NAMES},\right. \\ \text{FEATURES}\left[\texttt{featureID}\right] \leftarrow \left(F_e,\, F_n,\, \texttt{clampInterval}(F_t, t_n)\left[[t_n, t_{t_2})\right] \leftarrow \texttt{type},\, F_p,\, F_c\right), \\ \left.\text{GROUPS}\right)\end{array}}$$

Figure 2.22

$$\texttt{getTypes}\left(\left(G_e,\, G_t,\, G_p,\, G_c\right),\, [t_n, t_m)\right) = G_t\left[[t_n, t_m)\right]$$
$$\texttt{getTypes}\left(\left(F_e,\, F_n,\, F_t,\, F_p,\, F_c\right),\, [t_n, t_m)\right) = G_t\left[[t_n, t_m)\right]$$

Figure 2.23

2.3.8 Change feature variation type rule

The rule in figure 2.22 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 ($F_t\left[t_n\right]_{\leq} = \{[t_{t_1}, t_{t_2})\}$) 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 the next page is similar to the **changeFeature-VariationType** rule in figure 2.22, and shows the semantics of changing

(CHANGE-GROUP-VARIATION-TYPE)

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

Figure 2.24

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.

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 next 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.

(CHANGE-FEATURE-NAME)

$$F_n\,[t_n] = \{\texttt{oldName}\} \qquad F_n\,[t_n]_{\leq} = \{[t_{n_1}, t_{n_2})\}$$
$$\text{NAMES}\,[\texttt{name}]\,[[t_n, t_{n_2})] = \emptyset$$
$$\text{FEATURES}\,[\texttt{featureID}] = \left(F_e,\, F_n,\, F_t,\, F_p,\, F_c\right)$$

$$\textbf{changeFeatureName}\,(\texttt{featureID}, \texttt{name})\ \text{at}\ t_n \triangleright (\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$
$$\longrightarrow$$
$$\left((\text{NAMES}\,[\texttt{name}]\,[[t_n, t_{n_2})] \leftarrow \texttt{featureID})\,[\texttt{oldName}] \leftarrow \texttt{clampInterval}(\text{NAMES}\,[\texttt{oldName}]\,, t_n),\right.$$
$$\left.\text{FEATURES}\,[\texttt{featureID}] \leftarrow \left(F_e,\, \texttt{clampInterval}(F_n, t_n)\,[[t_n, t_{n_2})] \leftarrow \texttt{name},\, F_t,\, F_p,\, F_c\right),\right.$$
$$\left.\text{GROUPS}\right)$$

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

We use an inductive proof structure to prove soundness and modularity for the rule system (section 2.3 on page 15). The base case is a proof by construction, in which we define a sound temporal feature model. Using the induction hypothesis that the initial model is sound, we prove that each rule preserves soundness and operates within the previously defined scope (section 2.2 on page 13).

3.1 Soundness for temporal feature models

TODO: Argue that the rules are *enough* to preserve soundness. TODO: Show what needs to be proved for an initial model. "correct by construction". In a well-formed structure. Lemma: What is a sound structure. Soundness theorem: Preserves soundness, using the lemma. TODO: Proof for each rule should be lemma, and the whole system is a theorem or something:). Remember to number the lemmas TODO: The structure of the proof should be introduced first TODO: In the theorem, emphasize the structure of the proof (inductive) TODO: Between the proofs, add some text; how to read the proof, how to understand etc. Also give context (how the lemmas are used later)

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.

We must first define what it means for a temporal feature model to be sound. Essentially, it means that if we converted the temporal feature model into a sequence of time points associated with feature models, each feature model would be sound.

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.

Furthermore, a feature model is a tree structure and must not contain cycles. There is an additional requirement that groups with types ALTERNATIVE or OR must contain at least two child features, but this is not taken into account in this thesis.

These requirements can be translated into rules for temporal feature models $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$. We assume that the first time point in the plan is t_0.

TFMWF1 A temporal feature model has exactly one root feature. We assume that the root ID is `rootID`, and that $\text{FEATURES}\,[\texttt{rootID}] = \big(R_e,\, R_n,\, R_t,\, R_p,\, R_c\big)$. This also means that $R_e = \{[t_0, \infty)\}$ — the root always exists, and that $R_p = \emptyset$ — the root never has a parent group.

TFMWF2 The root feature must be mandatory. This means that

$$R_t = \{[\,[t_0, \infty) \mapsto \text{MANDATORY}\,]\}$$

where R_t is the types map of the root feature.

TFMWF3 At any time $t_n \geq t_0$, each feature has exactly one unique name, variation type and (potentially empty) collection of subgroups. Given a feature ID `featureID`, this means that if FEATURES $[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$ and $t_n \in_{\leq} F_e$, then

(i) $F_n[t_n] = \{\texttt{name}\}$ — the feature has exactly one name,

(ii) NAMES $[\texttt{name}][t_n] = \{\texttt{featureID}\}$ — the name is unique *at the time point* t_n,

(iii) $F_t[t_n] = \{\texttt{type}\}$ with $\texttt{type} \in \{\text{MANDATORY}, \text{OPTIONAL}\}$ — the feature has exactly one type, and

(iv) $F_c[t_n] = C$, such that $\bigcup C$ is a set of the group IDs, and if $\texttt{groupID} \in \bigcup C$ and GROUPS $[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$, then $G_p[t_n] = \{\texttt{featureID}\}$ — if a group is listed as a subgroup of a feature, then the feature is listed as the parent of the group at the same time.

TFMWF4 At any time $t_n \geq t_0$, each group has exactly one variation type. Given a group ID `groupID`, this means that if GROUPS $[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$ and $t_n \in_{\leq} G_e$, then $G_t[t_n] = \{\texttt{type}\}$ for $\texttt{type} \in \{\text{AND}, \text{OR}, \text{ALTERNATIVE}\}$.

TFMWF5 At any time $t_n \geq t_0$, each feature, except for the root feature, must be part of exactly one group. Formally, given a feature ID $\texttt{featureID} \neq \texttt{rootID}$, if FEATURES $[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, and $t_n \in_{\leq} F_e$, then $F_p[t_n] = \{\texttt{groupID}\}$ with GROUPS $[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$, $t_n \in_{\leq} G_e$, and $\texttt{featureID} \in \bigcup G_c[t_n]$.

TFMWF6 At any time $t_n \geq t_0$, each group must have exactly one parent feature. Formally, given a group ID `groupID`, if GROUPS $[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$ and $t_n \in_{\leq} G_e$, then $G_p[t_n] = \{\texttt{featureID}\}$, and FEATURES $[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$ with $\texttt{groupID} \in \bigcup F_c[t_n]$.

TFMWF7 At any time t_n, a group with types ALTERNATIVE or OR must not contain MANDATORY features. Formally, given a group ID `groupID` with GROUPS $[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$, if $F_t[t_n] = \{\texttt{type}\}$ with $\texttt{type} \in \{\text{ALTERNATIVE}, \text{OR}\}$, and if $\texttt{featureID} \in \bigcup F_c[t_n]$ and FEATURES $[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, then $F_t[t_n] = \{\text{OPTIONAL}\}$.

We must add two additional requirements:

TFMWF8 For a feature with ID `featureID` such that FEATURES $[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, if $t_n \notin_{\leq} F_e$, then $F_n[t_n] = F_t[t_n] = F_p[t_n] = F_c[t_n] = \emptyset$, and for all keys `name` in NAMES, $\texttt{featureID} \notin$

NAMES [name] $[t_n]$ — no name belongs to the feature. Similarly, for a group with ID groupID such that GROUPS [groupID] $= (G_e, G_t, G_p, G_c)$, if $t_n \notin_{\leq} G_e$, then $G_t[t_n] = G_p[t_n] = G_c[t_n] = \emptyset$. In other words, a feature or a group which does not exist cannot have a name, a type, a parent, or a child.

TFMWF9 The temporal feature model contains no cycles, which means that at any time point $t_n \geq t_0$, for any feature or group that exists at t_n, if we follow the parent chain upwards, we never encounter the same feature or group twice. In other words, no feature or group is its own ancestor.

Together, these requirements form the basis of the soundness proofs. We assume that the original plan is sound, so each of these requirements is assumed to be true for the original temporal feature model. Furthermore, we prove that the requirements must still hold for the updated model if the rule can be applied.

Soundness of the rules In the following sections, we prove that each rule is sound, and conclude that the system is sound.

For each rule, the proof for soundness includes three parts:

(i) proving that the rule operates strictly within the previously defined temporal and spatial scopes (section 2.2 on page 13),

(ii) that the rule preserves well-formedness, as defined in the above requirements ***TFMWF1-9***, and

(iii) that the rule updates the model correctly, preserving soundness as well as respecting the semantics of the operation.

3.2 Soundness of the Add feature rule

TODO: Use a proof environment that clearly separate definitions from proofs. The proofs should contain syntactic correctness, not semantic (that part is moved out of the proof)

See figure 2.3 on page 16 for the ADD-FEATURE rule. Let

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

be the initial state. Recall that this operation adds the feature with ID featureID to the temporal feature model (NAMES, FEATURES, GROUPS)

from t_n to t_m. We assume that $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ is well-formed, as defined in ***TFMWF1-9***.

TODO: Show that the feature model outside the scope is unchanged

Scope Recall from section 2.2 on page 13 that the temporal scope of this operation is $[t_n, t_m)$, and the spatial scope is the feature itself, the parent group and the name.

In the rule, we look up only the feature ID, the parent group ID, and the name, and update only the name, feature, and parent group. Thus, the rule operates within the spatial scope of the operation. Furthermore, the only interval looked up in the interval maps and sets of the model is $[t_n, t_m)$, which is exactly the temporal scope of the rule. Hence the rule operates strictly within the temporal and spatial scopes of the operation.

Lemma 3.1. The ADD-FEATURE rule operates strictly within the scope of the **addFeature** operation.

Preserving well-formedness If the rule is applied, the well-formedness requirements must hold for the updated feature model.

Since the rule checks that the feature does not already exist during the temporal scope, it is impossible that `featureID` $=$ `rootID`. Thus the rule does not affect the root feature, and ***TFMWF1*** and ***TFMWF2*** hold for the updated temporal feature model.

Because we assume that ***TFMWF8*** holds for the original model, and the feature does not exist during $[t_n, t_m)$, the feature has no name, type, or subgroups in the original plan. When we add the feature to the feature model using `setFeatureAttributes`, we give the feature exactly one name and one type during the temporal scope, and the set of child groups is empty. The temporal scope is also added to the feature's existence set, so only the new feature has the ID `featureID` during the temporal scope. To link the feature ID to the name, the rule sets the feature ID as the value at key `name` in the NAMES map during the temporal scope, so the name is unique during the temporal scope. Consequently, ***TFMWF3*** holds.

The rule does not modify the parent group's variation type, so ***TFMWF4*** is preserved in the modified temporal feature model.

Similarly to the argument for ***TFMWF3***, the parent group ID is uniquely defined for the feature in `setFeatureAttributes`, and `featureID` is added to the parent group's set of child features, so the new feature is part of exactly one group. Since we do not remove any other feature IDs from the

parent group's set of features, and as we already established that the new feature is not the root feature, ***TFMWF5*** is preserved.

The new feature does not have any subgroups during the temporal scope, and we do not modify the parent group's parent feature. Under the assumption that ***TFMWF6*** holds in the original model, it still holds after applying the ADD-FEATURE rule.

The rule verifies that all of the parent group's types are compatible with the added feature's type during the temporal scope, so ***TFMWF7*** holds after applying the rule.

Since the rule adds the temporal scope to the new feature's existence table, and since the parent group exists in the original plan, ***TFMWF8*** is preserved after the rule is applied.

It is furthermore impossible that adding this feature creates a cycle in the modified model. The new feature has no subgroups, so it cannot be part of a cycle. Because of the assumption that ***TFMWF9*** holds in the original plan, and applying the rule does not introduce a cycle, this requirement still holds.

As the rule operates within the scope (lemma 3.1 on the previous page), it does not affect any other part of the plan.

We conclude that the ADD-FEATURE rule preserves well-formedness for the temporal feature model, according to well-formedness rules ***TFMWF1-9***.

Lemma 3.2. The ADD-FEATURE rule preserves well-formedness of the temporal feature model.

TODO: Mention that the scope part ensures that no other part of the plan is affected

Correctness of model modification The operation is intended to add the feature with ID `featureID` to the temporal feature model during the interval $[t_n, t_m)$.

After adding the feature to the temporal feature model, looking up the name `name` in the NAMES map at interval key $[t_n, t_m)$ should give the value `featureID`. Indeed, since the NAMES map is updated thus:

$$\text{NAMES}\,[\texttt{name}]\,[[t_n, t_m)] \leftarrow \texttt{featureID}$$

, then due to the semantics of map assignment (definition 2.2 on page 7), and lookup in interval maps (definition 2.4 on page 8),

$$\text{NAMES}\,[\texttt{name}]\,[[t_n, t_m)] = \{\texttt{featureID}\}$$

will hold.

Similarly, if we wish to lookup information about the feature during the interval $[t_n, t_m)$ in the modified model, the results should match the information in the operation. The rule assigns

$$\texttt{setFeatureAttributes}(\text{FEATURES}\,[\texttt{featureID}]\,, [t_n, t_m), \texttt{name}, \texttt{type}, \texttt{parentGroupID})$$

to FEATURES $[\texttt{featureID}]$.

According to the semantics of assignment (section 2.1.2 on page 7) and `setFeatureAttributes` (figure 2.5 on page 17), and given that FEATURES $[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, then

$$[t_n, t_m) \in F_e \quad \text{the feature exists} \quad (1)$$

$$F_n\,[[t_n, t_m)] = \{\texttt{name}\} \quad \text{the feature has the expected name} \quad (2)$$

$$F_t\,[[t_n, t_m)] = \{\texttt{type}\} \quad \text{the feature has the expected type} \quad (3)$$

$$F_p\,[[t_n, t_m)] = \{\texttt{parentGroupID}\} \quad \text{the feature has the expected parent group} \quad (4)$$

$$F_c\,[[t_n, t_m)] = \emptyset \quad \text{the feature has no children} \quad (5)$$

Statement (1) holds due to the line $F_e \cup [t_n, t_m)$ in `setFeatureAttributes`. The next four hold due to both premises in the rule and modifications in the function. Due to the premise $[t_n, t_m) \not\subseteq F_e$, which means that the feature does not previously exist at any point during the interval, and since ***TFMWF8*** is assumed to hold for the original model, the original feature does not have a name, type, parent group or child groups during the interval. In the function `setFeatureAttributes`, the name is added ($F_n\,[[t_{start}, t_{end})] \leftarrow \texttt{name}$), and so is the type ($F_t\,[[t_{start}, t_{end})] \leftarrow \texttt{type}$) and the parent group ($F_t\,[[t_{start}, t_{end})] \leftarrow \texttt{parentGroupID}$). The child groups are not modified, and so (5) holds.

The child features of the group must also be updated according to the semantics of the operation. After applying the rule, given that GROUPS $[\texttt{parentGroupID}] = (G_e, G_t, G_p, G_c)$,

$$\texttt{featureID} \in \bigcup G_c\,[[t_n, t_m)]$$

, meaning that the feature is in the parent group's set of child features in the updated model. This holds because GROUPS $[\texttt{parentGroupID}]$ is assigned $\texttt{addChildFeature}(\text{GROUPS}\,[\texttt{parentGroupID}]\,, [t_n, t_m), \texttt{featureID})$ (see figure 2.6 on page 17), which modifies G_c by adding `featureID` to the set of child features at interval key $[t_n, t_m)$.

Lemma 3.3. The **ADD-FEATURE** rule updates the temporal feature model according to the semantics of the **addFeature** operation.

3.3 Soundness for the Add group rule

See figure 2.7 on page 18 for the **ADD-GROUP** rule. Let

$$\textbf{addGroup}(\texttt{groupID}, \texttt{type}, \texttt{parentFeatureID}) \text{ at } [t_n, t_m) \triangleright (\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$

be the initial state. Recall that this operation adds the group with ID `groupID` to the temporal feature model (NAMES, FEATURES, GROUPS) from t_n to t_m.

Scope Recall from section 2.2 on page 13 that the temporal scope of this operation is $[t_n, t_m)$, and the spatial scope is the group itself and the parent feature.

In the premise of the rule, only `groupID` and `parentFeatureID` are looked up in the temporal feature model. Consequently, the premise stays within the spatial scope of the rule. In the conclusion of the rule, FEATURES are assigned to and looked up at `parentFeatureID`, and GROUPS at `groupID`. The helper functions `addChildGroup` (figure 2.9 on page 18) and `setGroupAttributes` (figure 2.8 on page 18) do not take the temporal feature model as argument, and so only affects the parent feature and the group itself, respectively.

As for the temporal scope, the only interval looked up in the rule is $[t_n, t_m)$. Hence the rule operates only within the defined temporal scope.

Lemma 3.4. The **ADD-GROUP** rule operates strictly within the scope of the **addGroup** operation.

Preserving well-formedness If the **ADD-GROUP** rule is applied, the resulting temporal feature model must be well-formed according to to the well-formedness rules ***TFMWF1-9***.

The rule does not change the root feature's existence or type, so it does not violate ***TFMWF1*** or ***TFMWF2***. The NAMES map is left unchanged, and the only change made to a feature is to the parent feature, adding `groupID` to the set of child groups at $[t_n, t_m)$. The only feature modified is the parent feature, and only in its child groups map F_c. Since `parentFeatureID` is assigned to the group's parent feature table F_p at the same key $[t_n, t_m)$, ***TFMWF3*** holds.

Given that ***TFMWF8*** holds in the original model, and as the rule premise makes certain that the group does not already exist during the interval $[t_n, t_m)$, the group does not have any types, parent features, or child

features during the interval. When the rule is applied, the group is given exactly one type and parent feature, and $[t_n, t_m)$ is added to its existence set. Thus ***TFMWF4***, ***TFMWF6***, and ***TFMWF8*** hold.

As for ***TFMWF5***, this requirement holds trivially given that it holds in the original model. No feature is added or removed from any group in the ***Add-Group*** rule, so this condition is not affected and thus still holds.

Similarly, ***TFMWF7*** will hold in the altered model given that it holds in the original one, since the new group does not contain any features during the temporal scope. For the same reason, the rule does not create a cycle, and so ***TFMWF9*** is true for the altered model.

Lemma 3.5. The ADD-GROUP rule preserves well-formedness of the temporal feature model.

Correctness of model modification The operation is intended to add the group with ID `groupID` to the temporal feature model during the interval $[t_n, t_m)$. Since groups have no names, this operation should not affect the NAMES map. Indeed, the rule reflects this, as the map is not changed in the transition.

However, the operation does naturally add information to the GROUPS map, assigning

$$\texttt{setGroupAttributes}(\text{GROUPS}\,[\texttt{groupID}]\,, \texttt{type}, \texttt{parentFeatureID})$$

to GROUPS $[\texttt{groupID}]$.

Looking up the added group's ID in the modified model should return the information we put in the operation, and given GROUPS $[\texttt{groupID}] = (G_e,\, G_t,\, G_p,\, G_c)$, the following statements hold:

$$[t_n, t_m) \in G_e \qquad \text{the group exists} \qquad (1)$$

$$G_t\,[[t_n, t_m)] = \{\texttt{type}\} \qquad \text{the group has the expected type} \qquad (2)$$

$$G_p\,[[t_n, t_m)] = \{\texttt{parentGroupID}\} \qquad \text{the group has the expected parent feature} \qquad (3)$$

$$G_c\,[[t_n, t_m)] = \emptyset \qquad \text{the group has no children} \qquad (4)$$

Statement (1) holds due to the line $G_e \cup [t_n, t_m)$ in `setGroupAttributes` (figure 2.8 on page 18). Given the semantics of assignment, also statement (2) and (3) hold, as the type and parent feature ID are assigned to $G_t\,[[t_n, t_m)]$ and $G_p\,[[t_n, t_m)]$ respectively in `setGroupAttributes`. Given that ***TFMWF8*** is true for the original model, and since `setGroupAttributes` does not modify G_c, statement (4) is also true.

Furthermore, we would expect the group to be listed as a child group of the parent feature in the modified model, so given that $\text{FEATURES}\left[\texttt{parentFeatureID}\right] = \left(F_e, F_n, F_t, F_p, F_c\right)$, then

$$\texttt{groupID} \in \bigcup F_p\left[[t_n, t_m)\right]$$

In the **ADD-GROUP** rule, $\texttt{addChildGroup}\left(\text{FEATURES}\left[\texttt{parentFeatureID}\right], [t_n, t_m), \texttt{groupID}\right.$ is assigned to $\text{FEATURES}\left[\texttt{parentFeatureID}\right]$. The function `addChildGroup` (figure 2.9 on page 18) adds `groupID` to the set of child features at key $[t_n, t_m)$, so according to the semantics of $\overset{\cup}{\leftarrow}$, it is indeed true that the group is in the parent feature's set of child group in the temporal scope.

Lemma 3.6. The **ADD-GROUP** rule updates the temporal feature model according to the semantics of the **addGroup** operation.

3.4 Soundness of the Remove feature rule

See figure 2.10 on page 19 for the **REMOVE-FEATURE** rule. Let

$$\textbf{removeFeature}(\texttt{featureID}) \text{ at } t_n \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$

be the initial state. Recall that this operation removes the feature with ID `featureID` from the temporal feature model $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ at t_n. Furthermore, let $\text{FEATURES}\left[\texttt{featureID}\right] = \left(F_e, F_n, F_t, F_p, F_c\right)$, and $[t_i, t_j) \in F_e$, with $t_i \leq t_n < t_j$. This means that the original plan added the feature at time t_i and removed it at t_j, with the possibility that $t_j = \infty$. In the latter case, there was no plan to remove the feature originally.

Scope As defined in section 2.2 on page 13, the **removeFeature** rule's temporal scope is $[t_n, t_k)$, where t_k is the time point in which the feature was originally planned to be removed. We can see from the description above that $t_k = t_j$; the end point in the feature's existence set containing t_n. We then have that the scope is defined as $[t_n, t_j)$. In the rule, we find the interval $[t_i, t_j)$ by looking up

$$F_e\left[t_n\right]_{\leq} = \{[t_{e_1}, t_{e_2})\}$$

According to the semantics of $\text{IS}\left[t_n\right]_{\leq}$, it is true then that $[t_i, t_j) = [t_{e_1}, t_{e_2})$, and so the temporal scope of the rule is $[t_n, t_{e_2}) = [t_n, t_j)$. Clearly, all time points looked up in the premise of the rule are contained within this interval, but the conclusion requires further examination. The NAMES map is assigned $\texttt{clampInterval}\left(\text{NAMES}\left[\texttt{name}\right], t_n\right)$ at key `name`. In

`clampInterval` (figure 2.11 on page 20), the interval $[t_{n_1}, t_{n_2})$ containing t_n in NAMES [`name`] is looked up and shortened to end at t_n instead of t_{n_2}. This modification stays within the scope of the temporal feature model, since the interval affected here is $[t_n, t_{n_2})$, and necessarily, $t_{n_2} \leq t_j$, since the feature cannot possibly have a name after it is removed according to ***TFMWF8***.

The FEATURES map is modified at key `featureID` by assigning `clampFeature`(FEATURES [`featu`
In `clampFeature` (figure 2.14 on page 20), the intervals of the feature's name, type, and parent are clamped to end at t_n. These modifications, too, stay within the temporal scope, for the reason explained in the above paragraph. The existence interval is clamped in a similar way, and so stays within the temporal scope as well.

Also, the GROUPS map is assigned `removeFeatureAt`(GROUPS [`parentGroupID`] , `featureID`, t_n)
at key `parentGroupID`. This helper function (figure 2.16 on page 20) modifies the parent group's table of subfeatures by calling `clampIntervalValue`(G_c, t_c, `featureID`), which behaves similarly to `clampInterval` by clamping the interval containing t_n. The difference is that it removes only `featureID` from the set of child groups, and adds the feature to the set of subfeatures at the shortened interval. We conclude that this modification, too, happens within the temporal scope of the operation, as looking up any time point outside of the temporal scope will return the same results as the original plan.

Recall that the spatial scope of the rule is the feature itself, its parent group, and its subtree. The premise

$$F_c\left[[t_n, t_{e_2})\right] = \emptyset$$

ensures that the operation is not applied unless the feature's subtree is empty. The only features and groups looked up is the feature itself and its parent group. Thus, the rule stays within the spatial scope.
Lemma 3.7. The **REMOVE-FEATURE** rule operates strictly within the scope of the **removeFeature** operation.

Preserving well-formedness The **REMOVE-FEATURE** rule contains the premise

$$F_p\left[[t_n, t_{e_2})\right] = \{\texttt{parentGroupID}\}$$

, ensuring that the feature has *exactly* one parent group during the temporal scope of the rule. Under the assumption that ***TFMWF1*** holds in the original model, the feature being removed cannot be the root feature, since the root has no parent group. Furthermore, it means that the feature does not move during the temporal scope, which would be a conflict. Therefore, both ***TFMWF1*** and ***TFMWF2*** hold in the modified model.

For any time point t_n in the temporal scope of the rule, $t_n \notin_{\leq} F_e$ due to the semantics of `removeFeatureAt` (figure 2.16 on page 20), so ***TFMWF3***

and ***TFMWF5*** hold trivially. As the only change made to a group is by the function `clampFeature` (figure 2.14 on page 20), and since that function does not modify the types map G_t of the group, ***TFMWF4*** holds given that it is true for the original model.

The premise $F_c\left[[t_n, t_{e_2})\right] = \emptyset$ ensures that the feature to be removed does not have any subgroups during the temporal scope, so no group is left without a parent in the updated model. Thus ***TFMWF6*** holds.

Suppose that the parent group has the type ALTERNATIVE or OR at some point during the temporal scope. In the original model, no child feature of the group has type MANDATORY due to the assumption that ***TFMWF7*** is true. The **REMOVE-FEATURE** rule does not add any features or change a feature type, so this requirement still holds for the modified model.

After applying the rule, we have that $[t_n, t_j) \not\trianglelefteq F_e$, which means that the feature does not exist during the temporal scope of the operation. To fulfil ***TFMWF8***, we must furthermore have that $F_n\left[[t_n, t_j)\right] = F_t\left[[t_n, t_j)\right] = F_p\left[[t_n, t_j)\right] = F_c\left[[t_n, t_j)\right] = \emptyset$, and that $\texttt{featureID} \notin \text{NAMES}\left[\texttt{name}\right]\left[[t_n, t_j)\right]$. The former statement holds due to the semantics of `clampFeature` and `clampInterval`; the feature's attributes are all clamped to end at the time of removal, and the premises on the form $F_x\left[[t_n, t_{e_2})\right] = \{v\}$ ensure that no changes are made to those attributes during the temporal scope. $F_c\left[[t_n, t_j)\right] = \emptyset$ is a premise in the rule (since $t_j = t_{e_2}$). As for the NAMES map, the mapping from `name` to $\left[\,[t_i, t_j) \mapsto \texttt{featureID}\,\right]$ is replaced by $\left[\,[t_i, t_n) \mapsto \texttt{featureID}\,\right]$ in the function $\texttt{clampInterval}(\text{NAMES}\left[\texttt{name}\right], t_n)$. Hence ***TFMWF8*** is true for the altered temporal feature model.

Under the assumption that no cycles exist in the original model, removing a feature does not create a new one, so ***TFMWF9*** is holds for the modified model as well.

Lemma 3.8. The **REMOVE-FEATURE** rule preserves well-formedness of the temporal feature model.

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.