An analysis method 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

An analysis method for modular soundness checking of feature model evolution plans

Ida Sandberg Motzfeldt

An analysis method for modular soundness checking of feature model evolution plans

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

Printed: Reprosentralen, University of Oslo

Abstract

A software product line (SPL) is a family of closely related software systems which capitalizes on the reusability and variability of the software products. An SPL can be modeled using a feature model, a tree-like structure from which all the configurations of the SPL can be derived. Large projects such as an SPL require long-term planning, and plans for SPLs may also be defined in terms of feature models, called feature model evolution plans (FMEP). An FMEP gives information about what a feature model looks like at each stage of the plan.

As business requirements often change, FMEPs should support intermediate change. Such changes may cause paradoxes in an FMEP, e.g. a node left without a parent, making the plan impossible to realise. The complex nature of FMEPs makes detecting paradoxes by hand impractical. Current tools exist to validate FMEPs, but require analysis of the entire plan even when a modification affects only small parts of it. For larger FMEPs, this is inefficient. Thus, there is a need for a method which detects such paradoxes in a more efficient way.

In this thesis, we present a representation for FMEPs, called an interval-based feature model (IBFM). This representation enables local validation, by which we mean validating only the parts of the plan that are affected by the change. We further define operations for updating an IBFM, and methods for detecting paradoxes resulting from an operation. Moreover, we give a proof of correctness for the method and an implementation as proof of concept.

Using these methods, it is possible to create an efficient soundness checker for modification of FMEPs. This may be used as basis for an SPL planning tool and will contribute to productivity.

Contents

List of Figures

Part I

Introduction and Background

Chapter 1

Introduction

A software product line (SPL) capitalizes on the similarity and variability of closely related software products [1]. The similarities and variability are captured by features, which are customer-visible characteristics of a system [1]. Each product in the product line (called a *variant*) comprises a selection of these features, resulting in a flexible and customizable set of variants available to customers. To model an SPL it is common to use a feature model, a tree-like structure with nodes representing features. From this model, a variant can be derived by selecting features. The feature model's structure creates restrictions for which variants are allowed, while they also make it possible to model all possible variants at once [2].

SPLs grow large as they are more profitable the more variants they originate [1], and evolve over time as requirements change [3, 4]. Complex projects require planning [5]. Intuitively, this means describing how the feature model should look at a future point in time. For instance, new technology may emerge that the manager wishes to incorporate in the product line, but which takes a year to implement. One can then plan how the feature model will look at that point, as well as at some earlier stages where the technology is partly included. However, as requirements change, plans must adapt, and it may be necessary to change an existing plan, for instance by removing or adding features. These retroactive changes can affect later parts of the plan, causing *paradoxes* that make the plan impossible to realise [6].

A simple example of a paradox can be seen in Figure 1.1 on the following page. Here, a feature model is illustrated as a normal tree for simplicity. In the original plan, a node A exists in time 1 and is removed at time 5. We modify the plan by adding a child node B to A at time 3. This change causes a paradox in time 5, since node B is left without a parent node. In this case, it would be simple to detect this paradox by hand, but given a plan with hundreds of nodes and points in time, paradoxes may be harder

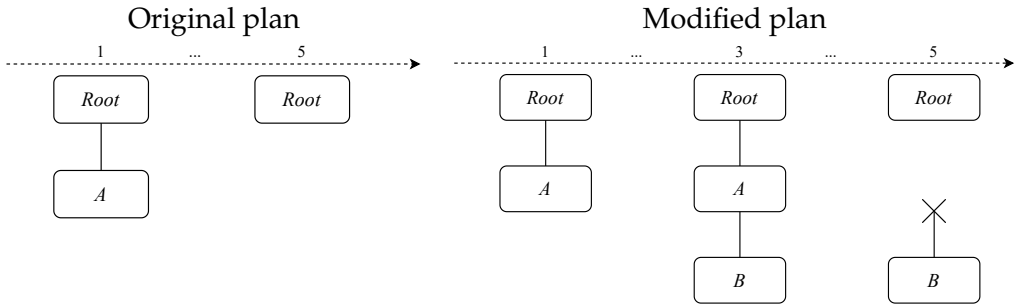


Figure 1.1: Simple paradox

to locate. Thus, there is a need for tooling that supports safe retroactive change to feature model evolution plans.

Notice also the difference between *feature model change*, i.e. planning to remove A at time 5, and *plan change*, i.e. modifying the original plan by introducing B at time 3. A plan may contain many changes to a feature model, but an evolution process will change the plans themselves. In this thesis we focus on plan changes.

1.1 The LTEP Project

This thesis is part of the LTEP research project, which was initiated in 2019 to address the lack of methodology and tooling for planning the long-term evolution of software product lines. It is a collaboration between the University of Oslo and the German university Technische Universität Braunschweig. The overarching goal of the project is to create methodology for the long-term evolution planning of SPLs, and we have published a paper [7] giving methods for verifying soundness of *feature model evolution plans* (FMEPs).

This method lets us detect paradoxes in a feature model evolution plan, and has been integrated into the SPL planning tool DarwinSPL[1] to make intermediate plan change possible; that is, modifying an earlier stage of the plan instead of adding to the latest stage. Such a change is exemplified in Figure 1.1, where the plan is changed by adding B at time 3. In the method created by LTEP, the process of changing the plan and verifying the change happens in the following way:

1) Introduce B at time 3
2) Derive the formal definition of the modified plan
3) Analyse the entire new plan

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

4) Locate the paradox that occurs at time 5, when we attempt to remove A even though it has a child node B.

This method requires us to analyse the entire plan each time a change to the plan is made, even though a change often does not affect much of the plan. In this example, only A is affected by the modification, and only between times 3 and 5. This thesis aims to remedy this by analysing *plan change* instead of entire plans, leveraging the knowledge that a change may only affect a small part of the plan. We can then exploit that adding B only affects its parent node A during the time between 3 and 5, ignoring the *Root* node and time 1. The added benefit in this example is small, but for larger plans, ignoring hundreds of features and points in time should gain us an advantage.

1.2 Research Questions

In order to reach the objective, we present research questions that will be addressed in the conclusion of the thesis.

RQ1 *Which operations are useful for modifying a feature model evolution plan?* In the LTEP project, we defined operations for modifying a feature model, but not a feature model evolution plan.

RQ2 *How can we capture all the states of a feature model evolution plan in such a way that the scope of each operation can be isolated?* Modifying a feature model evolution plan does not necessarily affect the entire plan. We wish to identify which parts of the plan *may* be affected by applying an operation, i.e. the *scope* of each operation. This problem requires a representation for feature model evolution plans that allows us to isolate the scope and analyse the effects of applying an operation *locally*.

RQ3 *How can we analyse change to ensure soundness?* Changing an intermediate stage of a feature model evolution plan may cause *paradoxes* — structural violations of the feature model — at a later stage of the plan. We aim to create an analysis method which ensures that any paradox arising from plan change is discovered and reported. This analysis method should be verifiably sound and possible to automate.

1.3 Contributions

In this thesis, we present a set of edit operations for changing feature model evolution plans. We furthermore define the scope of each of these operations, meaning that we deduce exactly which parts of a plan may be affected by each operation. A representation for feature model evolution plan is devised with the aim to easily isolate a scope for analysis. Based on the scope and representations, we create an analysis method for validation and application of the edit operations. The analysis is formalized as a set of rules, giving detailed specification of when an operation may be applied to the evolution plan, and how to apply the modification. We implement a prototype[2] of the analysis as proof of concept. Finally, we give an inductive proof that the rule set is sound by showing that each rule preserves soundness, that each rule stays within the specified scope, and that it updates the evolution plan correctly according to the semantics of the operation applied.

1.4 Chapter Overview

Chapter 2 gives background on software product lines, feature models, and feature model evolution plans, which form the basis of this thesis. Moreover, we give some background on static analysis, as the problems we deal with here bear similarity to those in static analysis.

Chapter 3 provides the definitions used throughout the thesis. These include the representation we use for feature model evolution plans — the interval-based feature model — as well as the operations we define for modifying them.

Chapter 4 defines rules for how to apply the operations to an interval-based feature model, and requirements for when an operation may be defined.

Chapter 5 details a proof for soundness of the rule system defined in Chapter 4 on page 33.

Chapter 6 describes the implementation. We present the implementation. We first give an overview of the types to provide intuition. We also briefly present the translation of the analysis rules, and give an example to show how it works.

Chapter 7 addresses our research questions, present possible improvements and future work, and concludes the thesis.

[2] TODO: Create a repo

Chapter 2

Background

In this chapter we provide some background on the topics relevant for this thesis. We begin by giving a general overview of software product lines, and continue by going into more detail on feature models and feature model evolution plans. Lastly, we give a short introduction to static analysis and its uses, as well as how it relates to the contributions of this thesis.

2.1 Software Product Lines

Software product lines (SPLs) are an engineering methodology used for developing products that share common features but have differences, for instance a product line of smartphones. When developing a software product line, engineers attempt to capitalize on the commonality by reusing components of source code for several of the products. For instance, all smartphones have technology for internet access. The final products of a software product line are called *variants*, which consist of a combination of features available in the SPL. In a smartphone SPL, a variant is a complete smartphone. When software product lines were still a novel concept, engineers tended to throw together variants by copying and pasting the components where needed. When the SPLs grew larger, this process became increasingly error-prone. Each time a component needed to be updated, all variants using the component must be reviewed. In later years, however, several technologies have emerged that exploit the reuse of the components, combining the components together into a final product. This makes maintaining code much more efficient and less error-prone, as each component only exists in one place. [1]

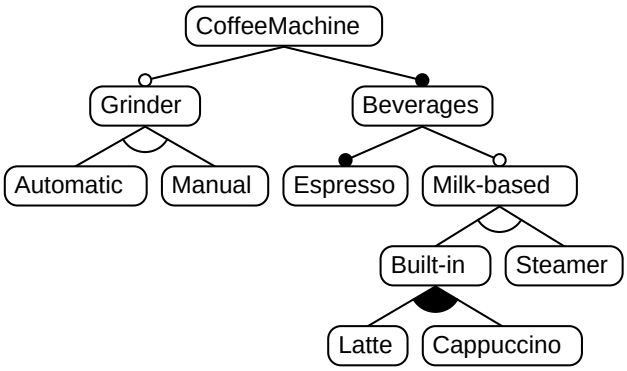


Figure 2.1: Example feature model for a coffee machine [1]

2.1.1 Feature Models

Variability management is the process of deciding which variants should be allowed, i.e. which combinations of features can be combined into a variant [8]. Formerly this was done as an informal process, often using spreadsheets and the engineers' intuition. To simplify and formalize this process we use *feature models* to model the relations between the features (which combinations are allowed, which are common to all variants, etc.). Feature models can also be used as documentation for a SPL, providing a common language between stakeholders [1]. A feature model is a tree-like structure where the nodes are features and groups of features. See Figure 2.1 for an example of a feature model. A feature cannot be selected in a variant unless its parent feature is also selected [2].

A group gives logical structure to the features, restricting the allowed combinations of the features. For instance, in an ALTERNATIVE group, exactly one of the features must be selected in every variant. In the example, the group under Grinder has this type. In a variant, a grinder cannot be both automatic and manual. Moreover, the features have types (OPTIONAL and MANDATORY). A MANDATORY feature must be selected in all variants, whereas an OPTIONAL feature may be left out. The black dot above Beverages means that this feature is mandatory, so all coffee machines provide beverages. Furthermore, since its subfeature Espresso is also mandatory, all coffee machines have espresso. However, only some coffee machines have milk-based drinks, as shown by the white dot above the Milk-Based feature. If selected, then either built-in drinks such as latte or cappuccino must be included in the variant, or the machine must have a steamer so the user can make milk-based drinks themselves. The group under Built-in is filled-in with black, which means that it is an OR group. In a variant where Built-in is chosen, one or both of Latte and Cappuccino must also be chosen, but not zero. The groups

[1] Created using DarwinSPL: `https://gitlab.com/DarwinSPL/DarwinSPL`

which are neither ALTERNATIVE nor OR have the type AND, which means that zero, one, or more of its subfeatures may be selected in a variant. There are several restrictions to the structure of a feature model. For instance, an ALTERNATIVE or OR group cannot contain a MANDATORY feature. Although all features have a type, not all of them are displayed in this figure. The root feature (here CoffeeMachine) must have type MANDATORY, since naturally it must be selected in all variants. Since an ALTERNATIVE or OR group cannot contain a MANDATORY feature, all features in those groups have the type OPTIONAL.

Feature models often also allow *cross-tree constraints*. These are similar to the parent-child relation in the feature model but are independent of the tree structure. For instance, one could imagine that the producer would always include an automatic grinder if the Built-In feature is selected, because the built-in feature does not work unless the machine grinds the coffee automatically. This cross-tree constraint could be expressed as "Built-In requires Grinder". Although cross-tree constraints are commonly used, we disregard them in this thesis. This is due to the fact that the cross-tree constraints add unwanted complexity, and we wish to exploit the tree structure of a feature model.

The formal structural requirements (well-formedness rules) to a feature model, as specified in [7] 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.

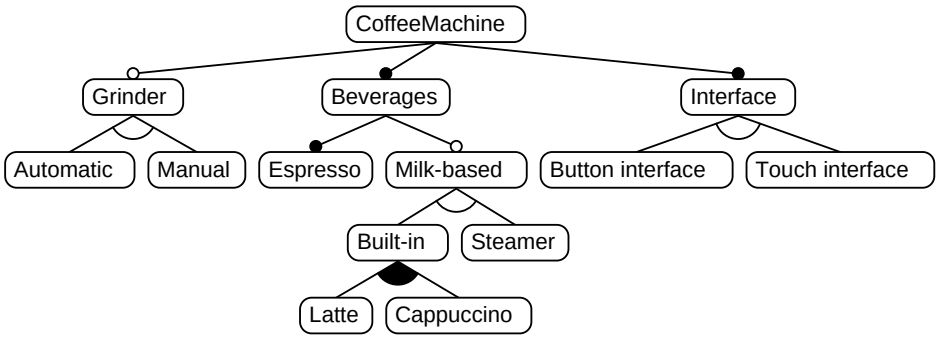


Figure 2.2: Coffee machine with added touch interface [2]

2.1.2 Feature Model Evolution Plans

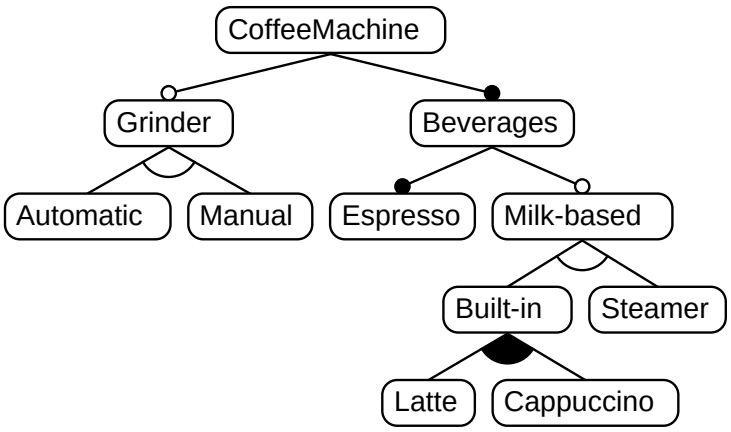
The evolution of an SPL can be planned using a *feature model evolution plan*. Software product lines often grow very large, and it is crucial to plan ahead. There exist tools for evolution planning, such as DarwinSPL [4]. An intuitive way to think of an evolution plan is a sequence of feature models associated with the points in time when they are planned to be realized. For instance, imagine that the first coffee machines in the software product line are represented by the feature model in Figure 2.1 on page 8. They are controlled by buttons, but as touch interfaces become more common, the manager decides to add coffee machines with a touch screen. This modification is included in Figure 2.2.

In the LTEP research project, we have formalized evolution plans as an initial model combined with a list of *time points*, which are defined as points in time, associated with edit operations, e.g. change type of "Beverages" to OPTIONAL. In the formalized feature model, each feature and group has a unique ID, and the edit operations use these IDs to uniquely identify the features and groups to be added, removed, or modified. To illustrate the idea, a simplified formalization of the evolution plan is shown in Figure 2.3 on the facing page. The initial model associated with time t_0 is the one shown in Figure 2.1 on page 8. Applying the operations under t_1 results in the feature model shown in Figure 2.2. In the example, IDs are not used to improve readability, but in the formal definitions, such details are included in the operations and the feature model.

Furthermore, we have published a semantics for these edit operations along with a formal definition of a feature model, letting us define a *sound* plan. The semantics is formalized as a set of structural operational semantic rules, detailing exactly which conditions must be fulfilled for an operation to be applied, and the way to construct the resulting feature

[2] Created using DarwinSPL: `https://gitlab.com/DarwinSPL/DarwinSPL`

t_0



t_1 Add new MANDATORY feature "Interface" to "CoffeeMachine" AND group
Add new ALTERNATIVE group to "Interface"
Add new feature "Button interface" to "Interface" ALTERNATIVE group
Add new feature "Touch interface" to "Interface" ALTERNATIVE group

Figure 2.3: Formalized feature model evolution plan

model after applying it. A sound plan is an evolution plan in which applying each operation in order results in a structurally sound feature model for each step. Using this semantics on a modified plan allows for validation of change. When changing a feature model evolution plan, soundness of the updated plan can be checked by modifying the list of edit operations and checking that the resulting plan is sound using the semantics. However, this approach models change to *feature models*, whereas the goal of this thesis is to model and analyse change to a *feature model evolution plan*.

2.2 Static Analysis

Static analysis attempts to predict the behaviour of a program without executing it [9]. This is different from *dynamic* analysis, which analyses programs while executing [10]. Static analysis methods have various uses, including compilers, for instance for type checking, error detection [11], and optimizations; and lint tools, which detect possible errors the programmer is making while coding [12]. Furthermore, it is used to prove properties about programs, i.e. that the program behaviour matches the specification [10]. Due to the halting problem, an algorithm cannot in general decide exactly how a program behaves, but there exist several methods to safely approximate information. For instance, live variable analysis discovers which variables *may* still be "alive" (meaning used in

the future) at a certain point of the program. Live variable analysis is done for optimization of memory allocation for variables. If a variable is known not to be live, it is safe to overwrite it with another variable. If a variable *may* be live, it should not be overwritten. This makes live variable analysis a *may analysis*. It is also a backward analysis since information about whether a variable is used in the future is carried backwards. There also exist *must analyses* and *forward analyses*. A must analysis looks for the greatest solution of things that *must* be true. In a forward analysis, the information flows forward; meaning that what has happened earlier in the program influences the analysis at later points in the program [9].

Unlike programs, it is possible to get a full overview of a feature model evolution plan. It is always possible to find the correct answer given the question "Does feature A exist at time 5?". An operational representation finds the answer by applying operations to the initial model until time 5 is complete, and checks if feature A exists in the resulting feature model. For intuition on why this must be true, imagine a (correct) program where we know all statements are assignment. This program terminates for a certainty since there is no branching. The same goes for an operational feature model evolution plan. There are no conditional statements and thus no branching. Since we know that every operational feature model evolution plan "terminates", we avoid the halting problem which is at the core of all static analysis of programs, which must always over- or under-approximate a solution to be certain that the analysis terminates.

Since static analysis deals with undecidable problems (properties of programs), it would be unwise to apply the methods from static analysis directly to soundness checking of feature model evolution plans. However, the terminology and syntax can still be applied to a problem of a less complex nature. May and must analyses always deal with *scope*. When asking if a variable is live, we are also defining the scope of the variable, meaning which parts of the program the variable may be part of. Furthermore, the method for *defining* the static program analyses can be applied to other domains. For instance, it is common to define these analyses in terms of rules on the form

$$\frac{\mathit{Conditions}}{\mathit{State} \longrightarrow \mathit{State}'}$$

, where *State* is the context when the rule is applied, and the *Conditions* consists of propositions concerning the *State*. After the rule is applied, State' is the result, usually a modified version of *State*. For instance, the semantics of an **if**-statement may be defined by the rules

$$\frac{\Gamma\,[b] = \top}{\langle \Gamma,\ \textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ S' \rangle \longrightarrow \langle \Gamma,\ S \rangle}\ \textsc{If}_1 \qquad \frac{\Gamma\,[b] = \bot}{\langle \Gamma,\ \textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ S' \rangle \longrightarrow \langle \Gamma,\ S' \rangle}\ \textsc{If}_2$$

Here, Γ is the context treated as a map from variable names to values,

and $\Gamma\,[b]$ returns the value of b at the time when the statement is executed, which is either $\top$ (true) or $\bot$ (false). If the expression b is true, then the next statement to be executed is S. If not, then the next statement is S'. Notice that no rule defines the program behaviour if the value of b is neither $\top$ nor $\bot$. This means that anything other than $\top$ or $\bot$ is an error, and an implementation of the language will provide an error message for such a case. This is a useful property of these rules, as there are often many ways to write an incorrect program, and all of these can be captured by *not* fitting the correct cases.

The syntax-driven, unambiguous, and compact nature of such rules make them popular for formally defining type systems and analysis tools [9]. Here, they give both the behaviour of the if-statement (semantics) using the syntax of the language, and, implicitly, they provide a method for checking correctness of an if-statement. If the expression is neither true nor false, then the program is incorrect. In this thesis, we largely exploit this property of only defining the correct cases when giving rules for soundness analysis of modifying feature model evolution plans.

2.2.1 Soundness

A feature model evolution plan may be viewed as a sequence of feature models associated with time points. In this context, soundness of a feature model evolution plan means that all of the feature models in the plan uphold the structural requirements ***WF1–7*** given in section 2.1.1 on page 8. In a sound plan, no paradoxes occur; for instance, no two features have the same name at the same time, no groups with type ALTERNATIVE or AND contain features of type MANDATORY, etc. This can be verified automatically, as we did in [7].

Part II

Definitions, Analysis, and Soundness Proofs

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 3

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. TODO: First, explain what the challenge is with doing modular checking. Why is it difficult, why I pay attention to it, how does it affect the rules I have created? TODO: Use examples!!!

3.1 Definitions

TODO: remove subsections TODO: Begin this section by explaining what I need and then defining what I need

3.1.1 Interval-Based Feature Model

Definition 3.1 (Interval-based feature model)**.** An *interval-based feature model* is defined as a triple (NAMES, FEATURES, 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" [7]. 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 interval-based feature models.

3.1.2 Maps

Definition 3.2 (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:

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

The semantics of assignment is given by the following:

$$(\text{MAP} \cup \{[\, k \mapsto v \,]\})\,[k] \leftarrow v' = \text{MAP} \cup \{[\, k \mapsto v' \,]\}$$
$$\text{MAP}\,[k] \leftarrow v' = \text{MAP} \cup \{[\, k \mapsto v' \,]\} \quad \text{if } k \text{ is not a key in MAP}$$

Note that we do not mutate maps, but rather obtain a new map when assigning a value to a key.

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.

3.1.3 Intervals

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

Definition 3.3 (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}.

To allow us to use intervals that have no end, we define the time point ∞, such that $[1, \infty)$ is an interval that starts at 1 and never ends. For all time points t_n, we have that $t_n \leq \infty$.

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

3.1.4 Interval Maps

Definition 3.4 (Interval map)**.** An *interval map* is a map where the key is an interval (Definition 3.3).

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 IM contains the mapping $[\,[t_1, t_5) \mapsto v\,]$, all of the queries in Figure 3.1 will return $\{v\}$ (assuming that $t_1 < t_2 < \ldots < t_5$):

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

Figure 3.1: 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 $\text{IM}\,[t_n]^v_{\leq}$ 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 $[t_n, t_m)$.

Assigning a value v to an empty interval in a map IM returns the same map, i.e. it is a no-op. Formally,

$$\text{IM}\,[[t_n, t_n)] \leftarrow v = \text{IM}$$

Likewise, the empty mapping $[\,[t_n, t_n) \mapsto v\,]$ is ignored, such that

$$\text{IM} \cup \{[\,[t_n, t_n) \mapsto v\,]\} = \text{IM}$$

3.1.5 Interval Sets

Definition 3.5 (Interval set)**.** An *interval set* is a set of intervals (Definition 3.3 on the preceding page).

Given an interval set IS, $[t_n, t_m) \in \text{IS}$ 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 $[t_n, t_m) \in_{\leq} \text{IS}$ iff there exists some interval $[t_i, t_j) \in \text{IS}$ 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 $[t_n, t_m) \in_{\lessapprox} \text{IS}$ iff there exists some interval $[t_i, t_j) \in \text{IS}$ with $[t_n, t_m)$ overlapping $[t_i, t_j)$.

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

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

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

TODO: The three following sections should be grouped together in a logical context

3.1.6 Map Entries

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 interval-based 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 $[\,$`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 interval-based 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 $[\,$`groupID` $\mapsto$ *group entry*$\,]$. A group has a type, a parent feature, and zero or more child features. 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 an interval map with the group's types, G_p is an interval map with parent feature IDs, and G_c is an interval map with child feature ID set values, the interval keys possibly overlapping.

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

3.1.7 How to Use the Interval-Based Feature Model

To provide intuition, we give some examples of how to use the interval-based feature model.

If a group with ID `groupID` with GROUPS $[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$ has the type ALTERNATIVE at time t_2, then

$$G_t\,[t_2] = \{\text{ALTERNATIVE}\}$$

The result is a set due to the nature of the interval keys; t_2 is contained within some interval key in G_t.

Suppose we have a feature with ID `featureID` where FEATURES $[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$. To check whether the feature exists at the time point t_5, we look up the time point in the feature's existence set F_e. Recall that F_e is an interval set. Then

$$t_5 \in_{\leq} F_e$$

means that t_5 is contained within some interval in F_e, so the feature does exist at time t_5. We use the operator $\in_{\leq}$ because the elements in F_e are intervals, and we wish to know whether t_5 is contained within one of those intervals. To get the feature's parent group ID at time t_5, we look up the time point in the feature's parent map F_p:

$$F_p\,[t_5] = \{\texttt{parentGroupID}\}$$

This is exactly the same as how we previously used $G_t\,[t_2]$. The resulting set $\{\texttt{parentGroupID}\}$ means that the only parent group the feature has at time t_5 is `parentGroupID`. Since the model is assumed to be sound, it makes sense that a feature which exists has exactly one parent group. If the feature did not exist, it would not have a parent group. The result would then be

$$F_p\,[t_5] = \emptyset$$

Although a feature always has exactly one parent group if it exists, it may have several child groups. Recall that the child group map F_c has set values, meaning that the values are sets of group IDs. Furthermore, the keys may overlap, since a feature may have 3 groups from t_3 to t_6, but 1 in the interval $[t_4, t_6)$. Thus, to obtain the set of child groups at time t_5, we must take the union of the result after looking up t_5 in the child group map F_c.

$$\bigcup F_c\,[t_5] = \{\texttt{childGroup1}, \texttt{childGroup2}, \texttt{childGroup3}, \texttt{childGroup4}\}$$

If we did not take the union, we would get something like

$$F_c\,[t_5] = \{\{\texttt{childGroup1}, \texttt{childGroup2}, \texttt{childGroup3},\}\,\{\texttt{childGroup4}\}\}$$

This is why, later in the thesis, we see expressions like

$$\texttt{groupID} \in \bigcup F_c\,[t_n]$$

This expression means that the group with ID `groupID` is a child group of our feature at time t_n.

Furthermore, we sometimes wish to locate the time when something ends; for instance, when a feature stops existing. If we want to find out when our feature is next removed after t_5, we can look it up in the existence set:

$$F_e\,[t_5]_{\leq} = \{[t_2, \infty)\}$$

The result set means that the feature is added at t_2, and is never removed. The syntax looks exactly the same for interval maps. If we want to know when the feature is next moved (after t_5), we use the same operator with the parent group map:

$$F_p\,[t_5]_{\leq} = \{[t_3, t_6)\}$$

The feature was moved to its current parent group at t_3, and will be moved next at t_6.

We often want to know what is true for an *interval*, not just a time point. In particular, we may want to check that the feature does not exist during some interval, for instance $[t_0, t_2)$. We then use the negated overlapping member operator $\notin_{\lessapprox}$:

$$[t_0, t_2)\ \notin_{\lessapprox}\ F_e$$

This predicate is true if no intervals in the set F_e overlaps $[t_0, t_2)$. If we had that $[t_1, t_3) \in F_e$, the above predicate would be false, since both intervals contain the time point t_1.

3.1.8 Example Evolution Plan

A small example of an interval-based feature model can be found in Figure 3.2 on the next page. It contains three features and one group, and describes an interval-based 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.

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

Figure 3.2: Small interval-based feature model

3.1.9 Operations

We define *operations* to alter the interval-based 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 $[\texttt{featureID} \mapsto (\text{EXISTENCE}, \dots)]$ such that $[t_i, t_j) \in \text{EXISTENCE}$ 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 $[\texttt{groupID} \mapsto (\text{EXISTENCE}, \dots)]$ such that $[t_i, t_j) \in \text{EXISTENCE}$ 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.

TODO: move scope and operations above definitions

3.2 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 interval-based feature model is affected by the change. In other words, if we look at the interval-based 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 subgroups. If the feature has or will have a subgroup during the interval, then it cannot be removed. Otherwise, there are no conflicts. When modifying the interval-based 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 well as the feature and the group themselves. 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, the feature, and its previous name. If it already exists within the feature model during the interval, or if the feature does not exist, 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 3.2 on page 26, 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.

Chapter 4

A Rule System for Analysis of Plan Change

TODO: Instantiate rules with a concrete example for the rules and the scope

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

$$\textbf{(RULE-LABEL)}\qquad \frac{\text{Premises}}{S \longrightarrow S'}$$

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 interval-based 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 4.1 contains premises verifying that the feature does not already exist when we wish to add it.

4.1 Add Feature Rule

Figure 4.1 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

(ADD-FEATURE)

$$[t_n, t_m) \not\in_{\gtrless} F_e \qquad [t_n, t_m) \in_{\leq} G_e \qquad \text{NAMES}\,[\texttt{name}]\,[[t_n, t_m)] = \emptyset \qquad t_n < t_m$$
$$\text{FEATURES}\,[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$$
$$\text{GROUPS}\,[\texttt{parentGroupID}] = (G_e, G_t, G_p, G_c)$$
$$\forall \texttt{gt}_{\in G_t[[t_n,t_m)]}(\texttt{compatibleTypes}(\texttt{gt}, \texttt{type}))$$

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

Figure 4.1: The **ADD-FEATURE** SOS rule

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

Figure 4.2: `compatibleTypes`

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 ($\text{NAMES}\,[\texttt{name}]\,[[t_n, t_m)] = \emptyset$), and the interval must start before it ends ($t_n < t_m$). 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 4.2), `setFeatureAttributes` (Figure 4.3), and `addChildFeature` (Figure 4.4).

```
setFeatureAttributes((F_e, F_n, F_t, F_p, F_c), [t_start, t_end), name, type
                     , parentGroupID)
  = ( F_e ∪ [t_start, t_end)
    , F_n [[t_start, t_end)] ← name
    , F_t [[t_start, t_end)] ← type
    , F_p [[t_start, t_end)] ← parentGroupID
    , F_c )
```

Figure 4.3: `setFeatureAttributes`

$$\texttt{addChildFeature}\left(\left(G_e,\, G_t,\, G_p,\, G_c\right), [t_{start}, t_{end}), \texttt{fid}\right)$$
$$= \left(G_e, G_t, G_p, G_c\,[[t_{start}, t_{end})] \overset{\cup}{\leftarrow} \texttt{fid}\right)$$

Figure 4.4: `addChildFeature`

4.2 Add Group Rule

The rule in figure 4.5 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 ($[t_n, t_m) \notin_{\lessgtr} G_e$), 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. Lastly, the interval must fulfil the condition $t_n < t_m$, meaning that it starts before it ends.

(ADD-GROUP)

$$\frac{\begin{array}{c}[t_n, t_m) \notin_{\lessgtr} G_e \qquad [t_n, t_m) \in_{\leq} F_e \qquad t_n < t_m \\ \text{GROUPS}\,[\texttt{groupID}] = \left(G_e,\, G_t,\, G_p,\, G_c\right) \\ \text{FEATURES}\,[\texttt{parentFeatureID}] = \left(F_e,\, F_n,\, F_t,\, F_p,\, F_c\right)\end{array}}{\begin{array}{c}\textbf{addGroup}(\texttt{groupID}, \texttt{type}, \texttt{parentFeatureID}) \text{ at } [t_n, t_m) \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\ \longrightarrow \\ (\text{NAMES}, \\ \text{FEATURES}\,[\texttt{parentFeatureID}] \leftarrow \texttt{addChildGroup}\,(\text{FEATURES}\,[\texttt{parentFeatureID}]\,,\, [t_n, t_m),\, \texttt{groupID})\,, \\ \text{GROUPS}\,[\texttt{groupID}] \leftarrow \texttt{setGroupAttributes}(\text{GROUPS}\,[\texttt{groupID}]\,, \texttt{type}, \texttt{parentFeatureID}))\end{array}}$$

Figure 4.5: The **ADD-GROUP** SOS rule

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

Figure 4.6: `setGroupAttributes`

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

Figure 4.7: `addChildGroup`

4.3 Remove Feature Rule

Figure 4.8 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 interval-based feature model by clamping all the relevant intervals to t_n, i.e. shortening them to end at t_n.

(REMOVE-FEATURE)

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

Figure 4.8: The **REMOVE-FEATURE** SOS rule

4.4 Remove Group Rule

The **REMOVE-GROUP** rule in Figure 4.16 on page 38 describes the semantics of removing a group in an interval-based 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.

```
clampInterval(MAP, t_c)
  = let {[t_start, t_end)} ← MAP [t_c]_≤
        {v} ← MAP [t_c]
        MAP' ← MAP \ [t_start, t_end)
    in MAP' [[t_start, t_c)] ← v
```

Figure 4.9: `clampInterval`

```
clampIntervalValue(MAP, t_c, v)
  = let {[t_start, t_end)} ← MAP [t_c]^v_≤
        MAP' ← MAP \^v [t_start, t_end)
    in MAP' [[t_start, t_c)] ←^∪ v
```

Figure 4.10: `clampIntervalValue`

```
clampSetInterval(IS, t_c)
  = let {[t_start, t_end)} ← IS [t_c]_≤
        IS' ← IS \ [t_start, t_end)
    in IS' ∪ {[t_start, t_c)}
```

Figure 4.11: `clampIntervalSet`

```
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 4.12: `clampFeature`

```
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 4.13: `clampGroup`

```
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 4.14: `removeFeatureAt`

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

Figure 4.15: `removeGroupAt`

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 4.9 on the preceding page), `clampIntervalValue` (Figure 4.10 on the previous page), and `clampGroup` (Figure 4.13 TODO: pass på at dette er på riktig side) helper functions to update the interval-based 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 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)

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

Figure 4.16: The **REMOVE-GROUP** SOS rule

4.5 Move Feature Rule

See Figure 4.17 on the facing page for the semantics of the **moveFeature** operation. The premise $\neg$`createsCycle` refers to the algorithm described in Section 4.5.1 on the next 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)

$$\neg\texttt{createsCycle} \qquad F_p\left[t_n\right]_{\leq} = \left\{\left[t_{p_1}, t_{p_2}\right)\right\} \qquad F_p\left[\left[t_n, t_{p_2}\right)\right] = \{\texttt{oldParentID}\}$$

$$\forall [t_{f_1}, t_{f_2}) \in F_t\left[\left[t_n, t_{p_2}\right)\right]_{\lessapprox} \forall [t_{g_1}, t_{g_2}) \in G_t\left[\left\langle [t_{f_1}, t_{f_2})\right\rangle_{t_n}^{t_{p_2}}\right]_{\lessapprox}$$

$$\forall \texttt{ft} \in F_t\left[[t_{f_1}, t_{f_2})\right] \forall \texttt{gt} \in G_t\left[\left[t_{g_1}, t_{g_2}\right)\right] (\texttt{compatibleTypes}(\texttt{gt}, \texttt{ft}))$$

$$\text{FEATURES}\left[\texttt{featureID}\right] = \left(F_e, F_n, F_t, F_p, F_c\right)$$
$$\text{GROUPS}\left[\texttt{newParentID}\right] = \left(G_e, G_t, G_p, G_c\right)$$

$$\textbf{moveFeature}\left(\texttt{featureID}, \texttt{newParentID}\right) \text{ at } t_n \triangleright$$
$$(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$
$$\longrightarrow$$
$$(\text{NAMES},$$
$$\text{FEATURES}\left[\texttt{featureID}\right] \leftarrow \left(F_e, F_n, F_t, \texttt{clampInterval}(F_p, t_n)\left[\left[t_n, t_{p_2}\right)\right] \leftarrow \texttt{newParentID}, F_c\right),$$
$$\left(\text{GROUPS}\left[\texttt{oldParentID}\right]\right.$$
$$\leftarrow \texttt{removeFeatureAt}\left(\text{GROUPS}\left[\texttt{oldParentID}\right], \texttt{featureID}, t_n\right)\left.\right)\left[\texttt{newParentID}\right]$$
$$\leftarrow \texttt{addChildFeature}(\text{GROUPS}\left[\texttt{newParentID}\right], [t_n, t_{p_2}), \texttt{featureID})$$

Figure 4.17: The **MOVE-FEATURE** SOS rule

4.5.1 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 $\texttt{ancestors}(IBFM, \texttt{node}, \texttt{time})$ which takes the interval-based feature model, a node, and a time point and returns a list of `node`'s ancestors at time point `time`.

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

Next, find a list of critical nodes. Let $A_n = \texttt{ancestors}(IBFM, n, t_1) = [a_1, a_2, \ldots, SN, \ldots, r]$ and $A_{c_1} = \texttt{ancestors}(IBFM, c_1, t_1) = [c_2, c_3, \ldots, c_n, SN, \ldots, r]$ 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 $n \in \texttt{ancestors}(IBFM, k, t_k)$. Report that the move caused a cycle and terminate.

2. k is not in n's subtree, so this move is safe. Let $A_k = \texttt{ancestors}(IBFM, k, t_k) = [k_1, k_2, \ldots, k_n, SN', \ldots, r]$, 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 4.18: `ancestors`

The rule identifies the scope $[t_n, t_{p_2})$ by looking up the interval in F_p containing t_n, and looks up the ID of the original parent group $F_p\left[[t_n, t_{p_2})\right] = \{\texttt{oldParentID}\}$. The complex formula checks that the types of the feature and its new parent group are compatible at all times during the temporal scope.

In the conclusion of the rule, the feature's parent map is updated to express that the feature has a new parent during the temporal scope, and the parent groups' subfeature maps are updated similarly.

4.6 Move Group Rule

See Figure 4.19 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)

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

Figure 4.19: The **MOVE-GROUP** SOS rule

TODO: This is hard

4.7 Change Feature Variation Type Rule

The rule in Figure 4.20 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

(CHANGE-FEATURE-VARIATION-TYPE)

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

Figure 4.20: The **CHANGE-FEATURE-VARIATION-TYPE** SOS rule

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

Figure 4.21: `getTypes`

4.8 Change Group Variation Type Rule

The rule in Figure 4.22 on the facing page is similar to the **changeFeatureVariationType** rule in Figure 4.20, 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.

4.9 Change Feature Name

The semantics of changing the name of a feature are shown in the **CHANGE-FEATURE-NAME** rule in Figure 4.23 on the facing page. The old name and the next planned name change are identified on the first line ($F_n\left[t_n\right] = \{\texttt{oldName}\}$ and $F_n\left[t_n\right]_{\leq} = \{[t_{n_1}, t_{n_2})\}$ 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

(CHANGE-GROUP-VARIATION-TYPE)

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

Figure 4.22: The **CHANGE-GROUP-VARIATION-TYPE** SOS rule

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)

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

Figure 4.23: The **CHANGE-FEATURE-NAME** SOS rule

Chapter 5

Soundness

In this chapter, we prove soundness for the analysis rules.

TODO: Elaborate the proofs kept in this chapter TODO: Prepare the reader for what is to come. To not overwhelm the reader with proofs, the details can be found in the appendix. Here I will emphasize how I prove things, give concrete samples, and point to where the rest of it can be found.

We use an inductive proof structure to prove soundness and modularity for the rule system (Section 4 on page 33). The base case is a proof by construction, in which we define a sound interval-based 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 3.2 on page 28). We present only two of the proofs in this chapter, as the others are much the same. The rest of the proofs can be found in Appendix A on page 69.

5.1 Soundness for Interval-Based Feature Models

The interval-based 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 an interval-based feature model to be sound. Essentially, it means that if we converted the interval-

based feature model into a sequence of time points associated with feature models, each feature model would be sound.

The well-formedness requirements listed in Section 2.1.1 on page 8 can be translated into rules for interval-based feature models (NAMES, FEATURES, GROUPS). We assume that the first time point in the plan is t_0.

IBFM1 An interval-based feature model has exactly one root feature. We assume that the root ID is `rootID`, and that $\text{FEATURES}\,[\texttt{rootID}] = (R_e, R_n, R_t, R_p, R_c)$. 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.

IBFM2 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.

IBFM3 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 $\text{FEATURES}\,[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$ and $t_n \in_{\leq} F_e$, then

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

(ii) $\text{NAMES}\,[\texttt{name}]\,[t_n] = \{\texttt{featureID}\}$ — 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 $\text{GROUPS}\,[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$, 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.

IBFM4 At any time $t_n \geq t_0$, each group has exactly one variation type. Given a group ID `groupID`, this means that if $\text{GROUPS}\,[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$ and $t_n \in_{\leq} G_e$, then $G_t\,[t_n] = \{\texttt{type}\}$ for $\texttt{type} \in \{\text{AND}, \text{OR}, \text{ALTERNATIVE}\}$.

IBFM5 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 $\text{FEATURES}\,[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, and $t_n \in_{\leq} F_e$, then $F_p\,[t_n] = \{\texttt{groupID}\}$ with $\text{GROUPS}\,[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$, $t_n \in_{\leq} G_e$, and $\texttt{featureID} \in \bigcup G_c\,[t_n]$. Conversely, if $\texttt{featureID} \in \bigcup G_c\,[t_n]$, then $F_p\,[t_n] = \texttt{groupID}$.

IBFM6 At any time $t_n \geq t_0$, each group must have exactly one parent feature. Formally, given a group ID `groupID`, if $\text{GROUPS}\,[\texttt{groupID}] =$

(G_e, G_t, G_p, G_c) and $t_n \in_{\leq} G_e$, then $G_p[t_n] = \{\texttt{featureID}\}$, and $\text{FEATURES}[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$ with $\texttt{groupID} \in \bigcup F_c[t_n]$.

IBFM7 At any time t_n, a group with types ALTERNATIVE or OR must not contain MANDATORY features. Formally, given a group ID groupID with $\text{GROUPS}[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$, 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 $\text{FEATURES}[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, then $F_t[t_n] = \{\text{OPTIONAL}\}$.

We must add two additional requirements:

IBFM8 For a feature with ID featureID such that $\text{FEATURES}[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, 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, $\texttt{featureID} \notin \text{NAMES}[\texttt{name}][t_n]$ — no name belongs to the feature. Similarly, for a group with ID groupID such that $\text{GROUPS}[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$, 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.

IBFM9 The interval-based 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 interval-based feature model. Furthermore, we prove that the requirements must still hold for the updated model if the rule can be applied.

5.2 Soundness of Each Rule

In the following sections, we prove that each rule is sound, and conclude that the system is sound. We rely upon the above defined well-formedness requirements ***IBFM1–9*** to show this.

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 3.2 on page 28),

(ii) that the rule preserves well-formedness, as defined in the above requirements ***IBFM1–9***, and

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

These parts are concluded with a lemma for each rule, and the lemmas are finally used to show that the entire rule system is correct.

5.2.1 Soundness of the Add Feature Rule

See Figure 4.1 on page 34 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 interval-based feature model $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ from t_n to t_m. We assume that $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ is well-formed, as defined in ***IBFM1-9***.

Scope Recall from Section 3.2 on page 28 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 or assigned to 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 5.1. 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 $\texttt{featureID} = \texttt{rootID}$. Thus the rule does not affect the root feature, and ***IBFM1*** and ***IBFM2*** hold for the updated interval-based feature model.

Because we assume that ***IBFM8*** 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, ***IBFM3*** holds.

The rule does not modify the parent group's variation type, so ***IBFM4*** is preserved in the modified interval-based feature model.

Similarly to the argument for ***IBFM3***, 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, ***IBFM5*** 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 ***IBFM6*** 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 ***IBFM7*** 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, ***IBFM8*** 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 ***IBFM9*** 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 5.1 on the facing page), it does not affect any other part of the plan.

We conclude that the **ADD-FEATURE** rule preserves well-formedness for the interval-based feature model, according to well-formedness rules ***IBFM1-9***.

Lemma 5.2. The **ADD-FEATURE** rule preserves well-formedness of the interval-based feature model.

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

After adding the feature to the interval-based 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 3.2 on page 20), and lookup in interval maps (Definition 3.4 on page 21),

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

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 $\text{FEATURES}\,[\texttt{featureID}]$.

According to the semantics of assignment (Section 3.1.2 on page 20) and `setFeatureAttributes` (Figure 4.3 on page 34), and given that $\text{FEATURES}\,[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, then

$[t_n, t_m) \in F_e$	the feature exists	(1)
$F_n\,[[t_n, t_m)] = \{\texttt{name}\}$	the feature has the expected name	(2)
$F_t\,[[t_n, t_m)] = \{\texttt{type}\}$	the feature has the expected type	(3)
$F_p\,[[t_n, t_m)] = \{\texttt{parentGroupID}\}$	the feature has the expected parent group	(4)
$F_c\,[[t_n, t_m)] = \emptyset$	the feature has subgroups	(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 $[t_n, t_m) \not\subseteq_{\gtrless} F_e$, which means that the feature does not previously exist at any point during the interval, and since ***IBFM8*** 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 $\text{GROUPS}\,[\texttt{parentGroupID}] = (G_e, G_t, G_p, G_c)$,

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

, meaning that the feature is in the parent group's set of child features in the updated model. This holds because $\text{GROUPS}\,[\texttt{parentGroupID}]$ is assigned $\texttt{addChildFeature}(\text{GROUPS}\,[\texttt{parentGroupID}], [t_n, t_m), \texttt{featureID})$ (see Figure 4.4 on page 35), which modifies G_c by adding `featureID` to the set of child features at interval key $[t_n, t_m)$.

Lemma 5.3. The **ADD-FEATURE** rule updates the interval-based feature model according to the semantics of the **addFeature** operation.

5.2.2 Soundness of the Move Feature Rule

See Figure 4.17 on page 39 for the **MOVE-FEATURE** rule. Let

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

be the initial state, and

$$(\text{NAMES}', \text{FEATURES}', \text{GROUPS}')$$

be the result state after applying the **MOVE-FEATURE** rule. Recall that this operation moves the feature with ID `featureID` to the group with ID `newParentID`.

Scope Recall that the temporal scope of the move-feature rule is $[t_n, t_k)$ (Section 3.2 on page 28), where t_k is the time point at which the feature is originally planned to be moved or is removed. In the rule, this scope is identified by

$$F_p\,[t_n]_{\leq} = \{[t_{p_1}, t_{p_2})\}$$

Here, the time point t_n for moving the feature is looked up in the feature's parent map's set of interval keys, and the expected result is $\{[t_{p_1}, t_{p_2})\}$. This means that there is a mapping $[\,[t_{p_1}, t_{p_2}) \mapsto \texttt{parentGroupID}\,]$ in F_p, with `parentGroupID` being the ID of the feature's parent group at time t_n, and this group stops being the feature's parent at t_{p_2}. Thus the temporal scope of this operation is $[t_n, t_{p_2})$. The only interval looked up or assigned to in the rule is $[t_n, t_{p_2})$, but it is necessary to also look at the cycle detection algorithm in Section 4.5.1 on page 39, since this is also referenced in the

rule by $\neg$`createsCycle`. Here, t_{p_2} is called t_e, and the algorithm states that it only looks at time points between t_n and t_e. Thus the rule operates strictly within the temporal scope of the **moveFeature** operation.

The spatial scope for this operation is defined as the *ancestors which the feature and the target group do not have in common*. In other words, the *new* ancestors of the feature after applying the rule. In the rule itself, only the feature with ID `featureID` and its new parent group with ID `newParentID` are looked up. However, the cycle detection algorithm must also be considered. Here, the ancestors of both the feature and the group at t_n are looked up, the first ancestor they have in common identified, and the new ancestors are collected into a list. If one of them is moved before t_e, the list is updated. Hence the algorithm's spatial scope is indeed the feature's new ancestors and the feature itself, and so the rule operates within the defined spatial scope.

Lemma 5.4. The MOVE-FEATURE rule operates strictly within the scope of the **moveFeature** operation.

Preserving well-formedness Since the rule verifies that the feature has a parent group, the feature being moved is not the root. Thus ***IBFM1*** and ***IBFM2*** hold. The rule does not update the name, type or subgroups of the feature, so ***IBFM3*** is true for the updated model. Nor does it modify the target group's type or parent feature, so ***IBFM4***, ***IBFM6***, and ***IBFM7*** also hold.

The modification made to FEATURES$[\texttt{featureID}]$ is to the parent group map F_p by

$$F'_p = \texttt{clampInterval}(F_p,\, t_n)\,\big[[t_n, t_{p_2})\big] \leftarrow \texttt{newParentID}$$

As discussed in earlier sections (e.g. Section A.2 on page 71), `clampInterval` replaces a mapping $[\,[t_i, t_j) \mapsto v\,]$ by $[\,[t_i, t_n) \mapsto v\,]$, with $t_n \leq t_j$. Thus $\texttt{clampInterval}(F_p,\, t_n)$ has no parent group during the temporal scope. The subsequent assignment of `newParentID` to $[t_n, t_{p_2})$ ensures that the feature has exactly one parent group during the temporal scope. This relation is reflected in the GROUPS$'$ map, with

$$\begin{aligned}
&\text{GROUPS}' = \\
&\quad \big(\text{GROUPS}\,[\texttt{oldParentID}] \\
&\leftarrow\!\texttt{removeFeatureAt}\,\big(\text{GROUPS}\,[\texttt{oldParentID}]\,, \texttt{featureID}, t_n\big)\,\big)\,[\texttt{newParentID}] \\
&\leftarrow\!\texttt{addChildFeature}\big(\text{GROUPS}\,[\texttt{newParentID}]\,, [t_n, t_{p_2}), \texttt{featureID}\big)
\end{aligned}$$

The feature is added to the target group by `addChildFeature` during the interval $[t_n, t_{p_2})$, and removed from the original parent group by `removeFeatureAt`. Consequently ***IBFM5*** holds for the updated model.

By the premise

$$\forall [t_{f_1}, t_{f_2}) \in F_t \left[[t_n, t_{p_2}) \right]_{\lessapprox} \forall [t_{g_1}, t_{g_2}) \in G_t \left[\left\langle [t_{f_1}, t_{f_2}) \right\rangle_{t_n}^{t_{p_2}} \right]_{\lessapprox}$$
$$\forall \texttt{ft} \in F_t \left[[t_{f_1}, t_{f_2}) \right] \forall \texttt{gt} \in G_t \left[[t_{g_1}, t_{g_2}) \right] (\texttt{compatibleTypes}(\texttt{gt}, \texttt{ft}))$$

in the rule, the types of the feature and its new parent group are compatible. For each interval key in F_t overlapping the temporal scope, and for each interval key in G_t overlapping both the aforementioned interval *and* the temporal scope, it checks whether the types they map to are compatible. To fulfil this, each type the feature has during the temporal scope must be compatible with the type the parent group has at the same time. Thus ***IBFM7*** holds for the modified model.

As the rule does not alter the feature's existence set, ***IBFM8*** is preserved.

The intention of the cycle detection algorithm in Section 4.5.1 on page 39 is to uphold ***IBFM9***. Given the assumption that the original interval-based feature model contains no cycles, if the altered model contains a cycle then the **moveFeature** operation introduced it, and the feature being moved must be part of the cycle. This could only happen if the feature became part of its own subtree during the temporal scope, which means that at some point, the feature occurs in its own list of ancestors. The algorithm looks at the feature's *new* ancestors, meaning the ancestors that the feature does not have in the original plan, but does in the new one. It then checks that none of those ancestors are moved to the feature's subtree. Thus the rule preserves ***IBFM9***.

Lemma 5.5. The **MOVE-FEATURE** rule preserves well-formedness of the interval-based feature model.

Correctness of model modification The operation is intended to move the feature with ID `featureID` to the group with ID `newParentID` during the temporal scope $[t_n, t_{p_2})$. After applying the **MOVE-FEATURE** rule, the only differences between the original and modified interval-based feature model should be

(i) The feature's parent group should be `newParentID` during the temporal scope

(ii) The feature should not appear in the original parent group's set of subfeatures during the temporal scope

(iii) The feature should appear in the new parent group's set of subfeatures

Given the modified map of parent groups F'_p and the original map F_p, we have that

$$F'_p = \texttt{clampInterval}(F_p, t_n)\ \left[\left[t_n, t_{p_2}\right)\right] \leftarrow \texttt{newParentID}$$

This statement assigns `newParentID` to the temporal scope $[t_n, t_{p_2})$ after applying $\texttt{clampInterval}(F_p, t_n)$, meaning that the original parent mapping is shortened to end at t_n, and a new mapping $[\,[t_n, t_{p_2}) \mapsto \texttt{newParentID}\,]$ is inserted. By semantics of assignment, it is clear that for t_i with $t_n \leq t_i < t_{p_2}$, $F'_p\,[t_i] = \{\texttt{newParentID}\}$, which is the desired result and fulfils (i).

By Lemma 5.5 on the previous page and ***IBFM5***, (ii) and (iii) follow from (i). In other words, since the updated interval-based feature model is well-formed, and the feature's parent group during the temporal scope is `newParentID`, the feature is not in the original parent group's set of subfeatures during the temporal scope, and is in the new parent group's set of subfeatures.

Lemma 5.6. The **MOVE-FEATURE** rule updates the interval-based feature model according to the semantics of the **moveFeature** operation.

5.3 Soundness of the Rule System

Section 5.2 on page 47 shows that each the SOS rules operates within the scope of an operation. [1]

Theorem 5.7. The rule system supports local modification and verification of interval-based feature models.

Section 5.2 on page 47 shows that each of the SOS rules preserves well-formedness of the interval-based feature model.[2]

Theorem 5.8. The rule system for local modification of interval-based feature models is well-formed.

Section 5.2 on page 47 shows that each of the SOS rules behaves according to the semantics of the operation in question.[3]

Theorem 5.9. The rule system for local modification of interval-based feature models modifies the model correctly.

[1] See lemmas 5.1, A.1, A.4, A.7, 5.4, A.10, A.13, A.16, A.19
[2] See lemmas 5.2, A.2, A.5, A.8, 5.5, A.11, A.14, A.17, A.20
[3] See lemmas 5.3, A.3, A.6, A.9, 5.6, A.12, A.15, A.18, A.21

Chapter 6

Implementation

In this chapter, we present the implementation. We first give an overview of the types to display the structure of the implementation. An example is presented to show how it looks in practice. We also briefly present the translation of the analysis rules, and give an example of an application.

6.1 Overview

We have created a prototype for an implementation of the analysis rules. The implementation can be found on GitHub[1]. The implementation is not meant to be integrated directly into a tool, but serves as a proof of concept that our analysis method is realisable in practice. It can also serve as a guide for how to interpret the rules where they are unclear, if they are to be realised as part of a SPL planning tool.

The prototype is implemented in Haskell[2], which is a strongly typed, purely functional programming language. We chose the language since a functional language corresponds closely to the mathematical nature of our analysis rules. Moreover, Haskell has an implementation of interval maps[3] which are easily adapted to our purposes.

The most important modules are `Types`, `Validate`, and `Apply`. We give brief introductions to these modules in the following sections.

[1] TODO: github
[2] `https://www.haskell.org/`
[3] `https://hackage.haskell.org/package/IntervalMap`

6.1.1 Translation from Definitions to Types

In the `Types` module we define all the types used throughout the project, corresponding closely with our definitions (Section 3.1 on page 19). Our time points are implemented as an abstract data type `TimePoint`. The possible `TimePoints` are `TP n`, where `n` is an integer, or `Forever`, which corresponds to ∞. For all integers `n`, we have that `TP n < Forever`. Our notion of intervals are translated to an abstract data type `Validity`. Using

```
Validity (TP 3) (TP 5)
```

gives us the interval $[3, 5)$. Similarly, `Validity (TP 1) Forever` corresponds to the interval $[1, \infty)$.

We base our implementation of the interval maps on the Haskell module `IntervalMap`. To customise it to our needs, we name our representation `ValidityMap`, specifying that the keys are `Validitys`. The `IntervalMap` module provides several useful functions, such as `containing`, which takes an `IntervalMap` and a `TimePoint` and returns all the keys containing the given time point.

We further define the data type `IntervalBasedFeatureModel`, which takes the root ID of the IBFM, a `NameValidities` map, a `FeatureValidities` map, and a `GroupValidities` map. This corresponds closely to our interval-based feature model (NAMES, FEATURES, GROUPS) (see Definition 3.1 on page 19). The `NameValidities` map is a Haskell `Map`[4] from `Name`, which is a `String`, to `ValidityMap FeatureID`, where `FeatureID` is a wrapper type for `String`. Recall from Section 3.1.6 on page 22 that our NAMES map is a map from names to interval maps with feature ID values, which resembles our implementation. The `FeatureValidities` map has `FeatureID` keys and `FeatureValidity` values. The `FeatureValidity` resembles our feature entries $(F_e, F_n, F_t, F_p, F_c)$. The interval set F_e is represented by a `ValidityMap ()`. The special type `()` (*unit*) has only one value, namely `()`. This lets us treat the `ValidityMap ()` as an interval map or an interval set (where the interval keys are the elements of the set), depending on our needs. The names map F_n is represented by a `ValidityMap Name`, the types map F_t by a `ValidityMap FeatureType`, the parent group map F_p by a `ValidityMap GroupID`, and the child group map F_c by a `ValidityMap (Set`[5] `GroupID)`.

A group (G_e, G_t, G_p, G_c) is defined in much the same way, with a `ValidityMap ()` for its existence interval set G_e, a `ValidityMap GroupType`

[4] `https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html`

[5] `https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Set.html`

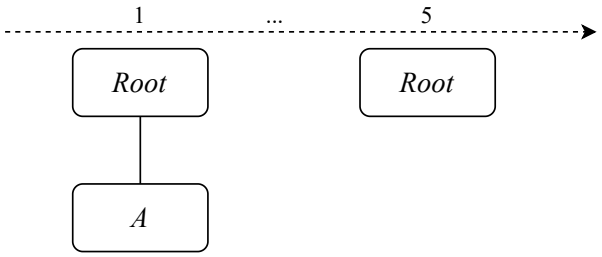


Figure 6.1: Simple plan

for its types map G_t, a `ValidityMap FeatureID` for its parent feature map G_p, and a `ValidityMap (Set FeatureID)` for its child feature map G_c.

The example in Figure 6.1 is formalized below in our previously defined representation, with only one feature (ID feature:root) and one group (ID group:A).

$$
\begin{aligned}
&(\{\,[\,\text{Root} \mapsto [\,[1,\infty) \mapsto \text{feature:root}\,]\,]\}\\
&\\
&,\{\,[\,\text{feature:root} \mapsto (\\
&\quad \{[1,\infty)\},\\
&\quad \{[\,[1,\infty) \mapsto \text{Root}\,]\},\\
&\quad \{[\,[1,\infty) \mapsto \text{MANDATORY}\,]\},\\
&\quad \emptyset,\\
&\quad \{[\,[1,5) \mapsto \text{group:A}\,]\})\,]\\
&\;\}\\
&\\
&,\;\{\,[\text{group:A} \mapsto (\\
&\quad \{[1,5)\},\\
&\quad \{[\,[1,5) \mapsto \text{AND}\,]\},\\
&\quad \{[\,[1,5) \mapsto \text{feature:root}\,]\},\\
&\quad \emptyset)]\\
&\;\})
\end{aligned}
$$

Below, the above example is translated to our Haskell representation[6].

```
im :: Validity -> a -> ValidityMap a
im = IM.singleton
```

[6]This example can be found in the GitHub repository, in `src/SimpleExample.hs`

```
simplePlan :: IntervalBasedFeatureModel
simplePlan =
  IntervalBasedFeatureModel
    (FeatureID "feature:root")
    [
      ( "Root"
      , im (Validity (TP 1) Forever) (FeatureID "feature:root")
      )
    ]
    [
      ( FeatureID "feature:root"
      , FeatureValidity
          (im (Validity (TP 1) Forever) ())
          (im (Validity (TP 1) Forever) "Root")
          (im (Validity (TP 1) Forever) Mandatory)
          mempty
          (im (Validity (TP 1) (TP 5)) [GroupID "group:A"])
      )
    ]
    [
      ( GroupID "group:A"
      , GroupValidity
          (im (Validity (TP 1) (TP 5)) ())
          (im (Validity (TP 1) (TP 5)) And)
          (im (Validity (TP 1) (TP 5)) (FeatureID "feature:root"))
          mempty
      )
    ]
```

The operations are interpreted quite directly, in two categories: The `AddOperations`, which take a `Validity` (interval) and an operation, and the `ChangeOperations`, which take a `TimePoint` and an operation. An operation in general is called a `TimeOperation`. To express **addFeature**(feature:B, B, MANDATORY, group:A) at $[3, \infty)$, we write

```
op =
   AddOperation (Validity (TP 3) Forever)
    (AddFeature (FeatureID "feature:B") "B" Mandatory
     ↪  (GroupID "group:A"))
```

6.1.2 Interpreting the Rules as Code

Each rule consists of a set of premises, and a conclusion which takes a state and returns a new state. In the implementation, we have chosen to

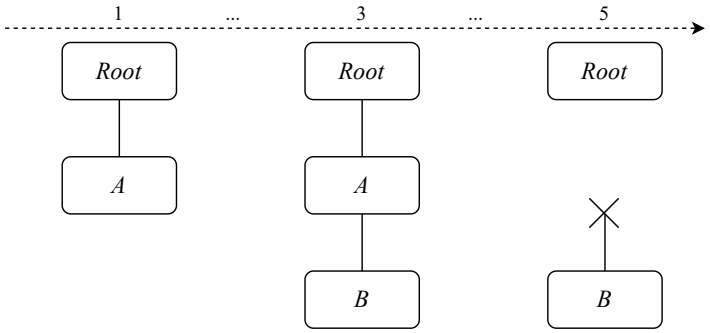


Figure 6.2: Illustration of the paradox

split these up, with one function for verifying the premises, and one for applying the operations.

The `Validate` module exports only the function `validate`, which takes a `TimeOperation` and an `IntervalBasedFeatureModel`. The most important difference between this function and the rules is that the function returns a list of errors if paradoxes occur. These errors belong to the type `ValidationError`, and consists of errors like `IncompatibleTypes`, to be returned if a feature and its parent group have incompatible types, `NameInUse`, if a feature is trying to use a name which already belongs to another feature, etc. This is done to show that the cause of a paradox can be located quite precisely.

To apply the operations, the `Apply` module exports the function `apply`, which takes a `TimeOperation` and an `IntervalBasedFeatureModel` and applies the operation to the model. This function works similarly to how it is defined in the rules.

The functions are combined in `validateAndApply`, which has the return type `Either [ValidationError] IntervalBasedFeatureModel`. If validation fails, it returns a list of errors, and if it succeeds, it applies the operation and returns the modified model. If we call `validateAndApply op simplePlan`, where `op` and `simplePlan` are defined above, we get the following result:

```
Left [ParentNotExists]
   :: Either [ValidationError] IntervalBasedFeatureModel
```

The list of errors, containing `ParentNotExists`, is wrapped in a `Left` to show that the type is a `ValidationError`. If the operation did not result in an error, we would get a result on the form `Right ibfm`, where `ibfm` is an `IntervalBasedFeatureModel`. The error here means that we are trying to add a feature, but the feature's parent does not exist at some point during the specified interval. In the eample, the parent group `group:A` is removed at 5, so the feature will be without a parent at that time. The paradox is

visualised in Figure 6.2 on the previous page.

Part III

Conclusion

Chapter 7

Conclusion and Future Work

In this chapter, we begin by reviewing the research questions, discussing to which degree we met our goals. Moreover, we suggest further improvements to our solution and future work. Lastly, we summarize our contributions in the conclusion.

7.1 Addressing the Research Questions

We presented our research question in the introduction, Section 1.2 on page 5. We now discuss how our solution addresses these questions.

RQ1 *Which operations are useful for modifying a feature model evolution plan?* We have chosen a set of edit operations (i.e. **addFeature(...)** at $[t_n, t_m)$) to edit a feature model evolution plan (see Section 3.1.9 on page 27). These operations allow us to add, remove, and move features and groups, as well as change their types and features' names. In other words, they can be used to alter all aspects of the features and groups, i.e., the *spatial* aspects of the feature model evolution plan. However, some alterations concerning *time* are not possible to achieve using these operations. For instance, if a feature exists from time t_1 to t_3, it is impossible to directly change the start of the feature's existence to t_0 without ending up with both the interval $[t_0, t_1)$ and $[t_1, t_3)$. However, we feel that the operations we have provided give a useful basis for a complete analysis tool for feature model evolution planning.

RQ2 *How can we capture all the states of a feature model evolution plan in such a way that the scope of each operation can be isolated?* The representation we have devised — the interval-based feature model, defined in Section 3.1.1 on page 19 — aims to address this research question. The interval-based feature model lets us look up the state of any

feature or group at any point in time given its ID, and the names used. This can be done without traversing the entire plan due to its use of maps, and the scope of each operation can be readily looked up. However, this solution gives quite a lot of redundancy, as all parent-child and feature-name relations are doubly present in the model. The redundancy makes updating the model more cumbersome, but it ensures that no operation requires a traversal of the entire plan. Ultimately, we feel that the trade-off is worth it, as the aim is to make sure that scopes can be isolated.

RQ3 *How can we analyse change to ensure soundness?* We have given rules for analysis of change in Section 4 on page 33. The rules rely on the assumption that the initial plan is sound, and as proven in Chapter 5 on page 45, they do guarantee that the resulting plan is sound if a rule can be applied. Furthermore, the prototype TODO: GitHub shows that the process can be automated. However, there are cases for which the rules may be too strict. Some sequences of operations may, applied in order, result in a sound plan, even though the plan is unsound after applying just one operation. For instance, suppose that a feature with ID `featureA` has the name "FeatureName" from time t_3 to t_4, but we wish for the feature with ID `featureB` to have the same name from t_1 to t_2. If we attempt to apply **changeFeatureName**(`featureB`, "FeatureName") at t_1, then the **ChangeFeatureName** rule will not apply, as `featureA` and `featureB` have the same name at time t_3. However, after applying **changeFeatureName**(`featureB`, "OldFeatureName") at t_2, the resulting plan would be sound. Nevertheless, this case is contrived and not likely to appear naturally.

7.2 Future Work

As mentioned in the previous section, there are aspects of our solution that can be improved upon. For an engineer to have complete freedom over her feature model evolution plan, one could create operations that let us extend and restrict the intervals in the interval-based feature model in all directions, not just for removal. A solution supporting batch operations — sequences of operations treated as a single operation — could also be useful in a complete implementation.

Hopefully, this analysis method can be applied in existing evolution planning tools such as DarwinSPL. For the benefits of modularity to shine, it should ideally be integrated tightly into the tool's representation. An implementation could also exploit the fact that this analysis has the potential for detailed error messages, since *change* is the subject of analysis,

and not the entire plan.

7.3 Conclusion

We have created a modular method for soundness analysis of change to feature model evolution plans. The analysis leverages the assumption of an initially sound evolution plan, and checks only those parts of the plan that *may* be affected by change. The representation we have created, the interval-based feature model, lets us isolate the scope of each operation, thus contributing to the modularity of the solution. We have given proofs of soundness and correctness for the analysis, and a prototype as proof of concept.

This analysis method may be implemented in an evolution planning tool and can help engineers to update evolution plans more securely and confidently.

Bibliography

[1] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005, ISBN: 978-3-540-24372-4. DOI: 10.1007/3-540-28901-1. [Online]. Available: https://doi.org/10.1007/3-540-28901-1.

[2] D. S. Batory, "Feature models, grammars, and propositional formulas", in *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, J. H. Obbink and K. Pohl, Eds., ser. Lecture Notes in Computer Science, vol. 3714, Springer, 2005, pp. 7–20. DOI: 10.1007/11554844_3. [Online]. Available: https://doi.org/10.1007/11554844%5C_3.

[3] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu, "Context-aware reconfiguration in evolving software product lines", *Sci. Comput. Program.*, vol. 163, pp. 139–159, 2018. DOI: 10.1016/j.scico.2018.05.002. [Online]. Available: https://doi.org/10.1016/j.scico.2018.05.002.

[4] M. Nieke, G. Engel, and C. Seidl, "Darwinspl: An integrated tool suite for modeling evolving context-aware software product lines", in *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2017, Eindhoven, Netherlands, February 1-3, 2017*, M. H. ter Beek, N. Siegmund, and I. Schaefer, Eds., ACM, 2017, pp. 92–99. DOI: 10.1145/3023956.3023962. [Online]. Available: https://doi.org/10.1145/3023956.3023962.

[5] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski, "Evofm: Feature-driven planning of product-line evolution", in *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering, PLEASE 2010, Cape Town, South Africa, May 2, 2010*, J. Rubin, G. Botterweck, M. Mezini, I. Maman, and A. Pleuss, Eds., ACM, 2010, pp. 24–31. DOI: 10.1145/1808937.1808941. [Online]. Available: https://doi.org/10.1145/1808937.1808941.

[6] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu, "Anomaly detection and explanation in context-aware software product lines", in *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume B, Sevilla, Spain, September 25-29, 2017*, M. H. ter Beek, W. Cazzola, O. Díaz, M. L. Rosa, R. E. Lopez-Herrejon, T.

Thüm, J. Troya, A. R. Cortés, and D. Benavides, Eds., ACM, 2017, pp. 18–21. DOI: 10.1145/3109729.3109752. [Online]. Available: https://doi.org/10.1145/3109729.3109752.

[7] A. Hoff, M. Nieke, C. Seidl, E. H. Sæther, I. S. Motzfeldt, C. C. Din, I. C. Yu, and I. Schaefer, "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*, R. E. Lopez-Herrejon, Ed., ACM, 2020, 8:1–8:12. DOI: 10.1145/3382025.3414964. [Online]. Available: https://doi.org/10.1145/3382025.3414964.

[8] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability management with feature models", *Sci. Comput. Program.*, vol. 53, no. 3, pp. 333–352, 2004. DOI: 10.1016/j.scico.2003.04.005. [Online]. Available: https://doi.org/10.1016/j.scico.2003.04.005.

[9] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999, ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6. [Online]. Available: https://doi.org/10.1007/978-3-662-03811-6.

[10] I. García-Ferreira, C. Laorden, I. Santos, and P. G. Bringas, "A survey on static analysis and model checking", in *International Joint Conference SOCO'14-CISIS'14-ICEUTE'14 - Bilbao, Spain, June 25th-27th, 2014, Proceedings*, J. G. de la Puerta, I. García-Ferreira, P. G. Bringas, F. Klett, A. Abraham, A. C. P. L. F. de Carvalho, Á. Herrero, B. Baruque, H. Quintián, and E. Corchado, Eds., ser. Advances in Intelligent Systems and Computing, vol. 299, Springer, 2014, pp. 443–452. DOI: 10.1007/978-3-319-07995-0_44. [Online]. Available: https://doi.org/10.1007/978-3-319-07995-0%5C_44.

[11] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools", *ACM Comput. Surv.*, vol. 44, no. 2, 6:1–6:42, 2012. DOI: 10.1145/2089125.2089126. [Online]. Available: https://doi.org/10.1145/2089125.2089126.

[12] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W. R. Marsh, "Industrial perspective on static analysis", *Softw. Eng. J.*, vol. 10, no. 2, pp. 69–75, 1995. DOI: 10.1049/sej.1995.0010. [Online]. Available: https://doi.org/10.1049/sej.1995.0010.

Appendix A

Remaining Soundness Proofs

A.1 Soundness for the Add Group Rule

See Figure 4.5 on page 35 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 interval-based feature model (NAMES, FEATURES, GROUPS) from t_n to t_m.

Scope Recall from Section 3.2 on page 28 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 interval-based 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 4.7 on page 36) and `setGroupAttributes` (Figure 4.6 on page 35) do not take the interval-based feature model as input, 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 A.1. 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 interval-based feature model must be well-formed according to to the well-formedness rules ***IBFM1–9***.

The rule does not change the root feature's existence or type, so it does not violate ***IBFM1*** or ***IBFM2***. 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 $[t_n, t_m)$, ***IBFM3*** holds.

Given that ***IBFM8*** 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 ***IBFM4***, ***IBFM6***, and ***IBFM8*** hold.

As for ***IBFM5***, 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, ***IBFM7*** 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 ***IBFM9*** is true for the altered model.

Lemma A.2. The ADD-GROUP rule preserves well-formedness of the interval-based feature model.

Correctness of model modification The operation is intended to add the group with ID `groupID` to the interval-based 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 $\text{GROUPS}\,[\texttt{groupID}]$.

Looking up the added group's ID in the modified model should return the information we put in the operation, and given $\text{GROUPS}\,[\texttt{groupID}] =$

(G_e, G_t, G_p, G_c), the following statements hold:

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

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

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

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

Statement (1) holds due to the line $G_e \cup [t_n, t_m)$ in `setGroupAttributes` (Figure 4.6 on page 35). 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 ***IBFM8*** 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 $\text{FEATURES}\,[\texttt{parentFeatureID}] = (F_e, F_n, F_t, F_p, F_c)$, then

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

In the **ADD-GROUP** rule,

$$\texttt{addChildGroup}(\text{FEATURES}\,[\texttt{parentFeatureID}], [t_n, t_m), \texttt{groupID})$$

is assigned to $\text{FEATURES}\,[\texttt{parentFeatureID}]$. The function `addChildGroup` (Figure 4.7 on page 36) 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 A.3. The **ADD-GROUP** rule updates the interval-based feature model according to the semantics of the **addGroup** operation.

A.2 Soundness of the Remove Feature Rule

See Figure 4.8 on page 36 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 interval-based feature model $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ at t_n. Furthermore, let $\text{FEATURES}\,[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, 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 3.2 on page 28, the **removeFeature** operation'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

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

According to the semantics of $\text{IS}\,[t_n]_{\leq}$, 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 $\texttt{clampInterval}(\text{NAMES}\,[\texttt{name}]\,, t_n)$ at key `name`. In `clampInterval` (Figure 4.9 on page 37), 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 interval-based 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 ***IBFM8***.

The FEATURES map is modified at key `featureID` by assigning

$$\texttt{clampFeature}(\text{FEATURES}\,[\texttt{featureID}]\,, t_n)$$

. In `clampFeature` (Figure 4.12 on page 37), 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

$$\texttt{removeFeatureAt}(\text{GROUPS}\,[\texttt{parentGroupID}]\,, \texttt{featureID}, t_n)$$

at key `parentGroupID`. This helper function (Figure 4.14 on page 37) modifies the parent group's set of subfeatures by calling

$$\texttt{clampIntervalValue}(G_c, t_c, \texttt{featureID})$$

, which behaves similarly to `clampInterval` by clamping the interval containing t_n. The difference is that it removes only `featureID` from the set of subfeatures, 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 subgroups. The premise

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

ensures that the operation is not applied unless the feature's set of subgroups 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 A.4. 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 ***IBFM1*** 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 ***IBFM1*** and ***IBFM2*** hold in the modified model.

For any time point t_n in the temporal scope of the rule, $t_n \not\in_{\leq} F_e$ due to the semantics of `clampFeature` (Figure 4.12 on page 37), so ***IBFM3*** holds trivially. The only change made to a group is by the function `removeFeatureAt` (Figure 4.14 on page 37), which removes the feature from the parent group's map of subgroups during $[t_n, t_j)$. Hence ***IBFM5*** holds, and since that function does not modify the types map G_t of the group, ***IBFM4*** 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 ***IBFM6*** 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 ***IBFM7*** 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 $[t_n, t_j) \not\in_{\lessgtr} F_e$, which means that the feature does not exist during the temporal scope of the operation. To fulfil ***IBFM8***, 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 $\texttt{featureID} \not\in \text{NAMES}\left[\texttt{name}\right]\left[[t_n, t_j)\right]$. 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 $[\,[t_i, t_j) \mapsto$ `featureID` $]$ is replaced by $[\,[t_i, t_n) \mapsto$ `featureID` $]$ in the function `clampInterval`$(\text{NAMES}\,[\texttt{name}]\,, t_n)$. Hence ***IBFM8*** is true for the altered interval-based feature model.

Under the assumption that no cycles exist in the original model, removing a feature does not create a new one, so ***IBFM9*** is holds for the modified model as well.

Lemma A.5. The **REMOVE-FEATURE** rule preserves well-formedness of the interval-based feature model.

Correctness of model modification The semantics of the **removeFeature** operation is that applying it should remove the feature from the plan from t_n until the point at which is was originally planned to be removed. Then if FEATURES$\,[\texttt{featureID}] = \left(F_e,\, F_n,\, F_t,\, F_p,\, F_c\right)$, and $F_e\,[t_n]_{\leq} = [t_i, t_j)$, then

$$
\begin{aligned}
[t_n, t_m) \notin_{\gtrless} F_e \quad & \text{the feature does not exist} && (1)\\
F_n\left[[t_n, t_m)\right] = \emptyset \quad & \text{the feature has no name} && (2)\\
F_t\left[[t_n, t_m)\right] = \emptyset \quad & \text{the feature has no type} && (3)\\
F_p\left[[t_n, t_m)\right] = \emptyset \quad & \text{the feature has no parent group} && (4)\\
F_c\left[[t_n, t_m)\right] = \emptyset \quad & \text{the feature has no subgroups} && (5)
\end{aligned}
$$

Since we established $[t_n, t_m) \notin_{\gtrless} F_e$ in the above paragraph, these statements follow directly from Lemma A.5 and ***IBFM8***. It further follows that no name is associated with `featureID` in the NAMES map, and that no group in the GROUPS map has the feature listed as a subfeature.

Lemma A.6. The **REMOVE-FEATURE** rule updates the interval-based feature model according to the semantics of the **removeFeature** operation.

A.3 Soundness of the Remove Group Rule

This proof is analogous to the one for the **REMOVE-FEATURE** rule. See Figure 4.16 on page 38 for the **REMOVE-GROUP** rule. Let

$$\textbf{removeGroup}(\texttt{groupID}) \text{ at } t_n \;\triangleright\; (\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$

be the initial state, and

$$(\text{NAMES}', \text{FEATURES}', \text{GROUPS}')$$

be the result state after applying the **REMOVE-GROUP** rule. Recall that this operation removes the group with ID `groupID` from the interval-based feature model (NAMES, FEATURES, GROUPS) at t_n. Furthermore, let GROUPS $[\texttt{groupID}] = (G_e, G_t, G_p, G_c)$, and $[t_i, t_j) \in G_e$, with $t_i \leq t_n < t_j$. This means that the original plan added the group 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 group originally.

Scope As defined in Section 3.2 on page 28, the **removeGroup** operation's temporal scope is $[t_n, t_k)$, where t_k is the time point in which the group was originally planned to be removed. We can see from the description above that $t_k = t_j$; the end point in the group'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

$$G_e\,[t_n]_{\leq} = \{[t_{e_1}, t_{e_2})\}\,.$$

According to the semantics of IS $[t_n]_{\leq}$, 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 untouched and thus outside the scope.

The GROUPS map is modified at key `groupID` by assigning

$$\texttt{clampGroup}(\text{GROUPS}\,[\texttt{groupID}]\,, t_n)$$

. In `clampGroup` (Figure 4.13 on page 37), the intervals of the group's type and parent feature are clamped to end at t_n. These modifications stay within the temporal scope, as $\texttt{clampInterval}(\text{MAP}, t_c)$ clamps the mapping with an interval key containing t_c to end at t_c. Due to ***IBFM8***, it is impossible that the group has a type or parent feature after t_j, which is the time point when the group was originally planned to be removed. Furthermore, the premise of the rule requires that $G_t\,[[t_n, t_j)] = \{\texttt{type}\}$ and $G_p\,[[t_n, t_j)] = \{\texttt{parentFeatureID}\}$, meaning that the group does not change its type or move during the temporal scope. Thus there is only one key in each of the group's type and parent feature maps containing t_n, and so the changed interval for these maps is $[t_n, t_j)$; the temporal scope. The existence interval is clamped in a similar way, and so stays within the temporal scope as well.

Also, the FEATURES map is assigned

$$\texttt{removeGroupAt}(\text{FEATURES}\,[\texttt{parentFeatureID}]\,, \texttt{groupID}, t_n)$$

at key `parentFeatureID`. This helper function (Figure 4.15 on page 37) modifies the parent feature's set of subgroups by calling

$$\texttt{clampIntervalValue}(F_c, t_c, \texttt{groupID})$$

, which behaves similarly to `clampInterval` by clamping the interval containing t_n. The difference is that it removes only `groupID` from the set of child groups, and adds the group to the set of subgroups 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 group itself, its parent feature, and its subfeatures. The premise

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

ensures that the operation is not applied unless the group's set of subfeatures is empty. The only features and groups looked up is the group itself and its parent feature. Thus, the rule stays within the spatial scope.

Lemma A.7. The **REMOVE-GROUP** rule operates strictly within the scope of the **removeGroup** operation.

Preserving well-formedness Let

$$\begin{aligned}
\text{GROUPS}\left[\texttt{groupID}\right] &= \left(G_e,\, G_t,\, G_p,\, G_c\right) \\
\text{GROUPS}'\left[\texttt{groupID}\right] &= \left(G'_e,\, G'_t,\, G'_p,\, G'_c\right) \\
\text{FEATURES}\left[\texttt{parentFeatureID}\right] &= \left(F_e,\, F_n,\, F_t,\, F_p,\, F_c\right) \\
\text{FEATURES}'\left[\texttt{parentFeatureID}\right] &= \left(F'_e,\, F'_n,\, F'_t,\, F'_p,\, F'_c\right)
\end{aligned}$$

The **REMOVE-GROUP** rule does not alter any feature's — in particular the root feature's — existence set or types map, and so ***IBFM1–2*** hold for the modified model. It does however modify the subgroup map F_c, applying `removeGroupAt` to the parent feature, the group's ID, and the removal time point. As previously argued, this function makes sure that the group ID is not in $\bigcup F'_c\left[[t_n, t_j)\right]$ — the parent feature's modified set of subgroups — so ***IBFM3*** holds.

Due to the semantics of `clampGroup` and `clampIntervalSet`, no time points in the temporal scope are contained in an interval in the modified group's existence set ($[t_n, t_j) \not\trianglelefteq G'_e$), so ***IBFM4***, ***IBFM6*** and ***IBFM7*** hold trivially.

Since a premise of the rule is

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

, the group does not have any subfeatures during the temporal scope in the original interval-based feature model. Due to the assumption that ***IBFM5*** is true for the original model, no feature has `groupID` listed as its

parent group, so no feature is left without a parent group when the group is removed from the temporal scope. It follows that ***IBFM5*** holds for the updated interval-based feature model as well.

As previously mentioned, $[t_n, t_j) \notin_{\leqq} G'_e$, meaning that the group does not exist during the temporal scope in the modified model. For ***IBFM8*** to hold for the updated model, we must then also have $G'_t\left[[t_n, t_j)\right] = G'_p\left[[t_n, t_j)\right] = G'_c\left[[t_n, t_j)\right] = \emptyset$. Recalling that $t_{e_2} = t_j$, then by definition of `clampGroup` and the premise $G_t\left[[t_n, t_{e_2})\right] = \{\texttt{type}\}$ in **ADD-GROUP**, we have that

$$\begin{aligned} G'_t &= \texttt{clampInterval}\left(G_t, t_n\right) \\ &= \texttt{clampInterval}\left(G''_t \cup [\,[t_{t_1}, t_j) \mapsto \texttt{type}\,], t_n\right) \\ &= G''_t \cup [\,[t_{t_1}, t_n) \mapsto \texttt{type}\,] \end{aligned}$$

Clearly, $G''_t\left[[t_n, t_j)\right] = \emptyset$. Furthermore, since $[t_{t_1}, t_n)$ does not overlap $[t_n, t_j)$, $G'_t\left[[t_n, t_j)\right] = \emptyset$. An analogous argument can be made for $G'_p\left[[t_n, t_j)\right] = \emptyset$. From the definition of `clampGroup`, $G_c = G'_c$, so by the premise $G_c\left[[t_n, t_j)\right] = \emptyset$ in the rule, $G'_c\left[[t_n, t_j)\right] = \emptyset$. Consequently ***IBFM8*** holds for the altered interval-based feature model.

Given that no cycles exist in the original model, removing a group does not create a new one, so ***IBFM9*** holds.

Lemma A.8. The **REMOVE-GROUP** rule preserves well-formedness of the interval-based feature model.

Correctness of model modification The semantics of the **removeGroup** operation dictate that the group should not exist, have a type, a parent, or subfeatures after being removed. A proof for this can be found in the previous paragraph. Moreover, the parent feature's map of child features should not contain `groupID` during the temporal scope. This is also proven in the previous paragraph. The NAMES map should not be modified. It is clear from the rule that NAMES = NAMES′, so this condition, too, is true.

Lemma A.9. The **REMOVE-GROUP** rule updates the interval-based feature model according to the semantics of the **removeGroup** operation.

A.4 Soundness of the Move Group Rule

See Figure 4.19 on page 41 for the **MOVE-GROUP** rule. Let

$$\begin{aligned} &\textbf{moveGroup}(\texttt{groupID}, \texttt{newParentID}) \text{ at } t_n \triangleright \\ &\qquad (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \end{aligned}$$

be the initial state, and

$$(\text{NAMES}', \text{FEATURES}', \text{GROUPS}')$$

be the result state after applying the **MOVE-GROUP** rule. Recall that this operation moves the group with ID `groupID` to the feature with ID `newParentID`.

Scope Recall that the temporal scope of the **MOVE-GROUP** rule is $[t_n, t_k)$ (Section 3.2 on page 28), where t_k is the time point at which the group is originally planned to be moved or is removed. In the rule, this scope is identified by

$$G_p\,[t_n]_{\leq} = \{[t_{p_1}, t_{p_2})\}$$

Here, the time point t_n for moving the group is looked up in the group's parent map's set of interval keys, and the expected result is $\{[t_{p_1}, t_{p_2})\}$. This means that there is a mapping $[\,[t_{p_1}, t_{p_2}) \mapsto \texttt{parentFeatureID}\,]$ in F_p, with `parentFeatureID` being the ID of the group's parent feature at time t_n, and this feature stops being the group's parent at t_{p_2}. Thus the temporal scope of this operation is $[t_n, t_{p_2})$. The only interval looked up or assigned to in the rule is $[t_n, t_{p_2})$, but it is necessary to also look at the cycle detection algorithm in Section 4.5.1 on page 39, since this is also referenced in the rule by $\neg$`createsCycle`. Here, t_{p_2} is called t_e, and the algorithm states that it only looks at time points between t_n and t_e. Thus the rule operates strictly within the temporal scope of the **moveGroup** operation.

The spatial scope for this operation is defined as the *ancestors which the group and the target feature do not have in common*. In other words, the *new* ancestors of the group after applying the rule. In the rule itself, only the group with ID `groupID` and its new parent feature with ID `newParentID` are looked up. However, the cycle detection algorithm must also be considered. Here, the ancestors of both the group and the feature at t_n are looked up, the first ancestor they have in common identified, and the new ancestors are collected into a list. If one of them is moved before t_e, the list is updated. Hence the algorithm's spatial scope is indeed the group's new ancestors and the group itself, and so the rule operates within the defined spatial scope.

Lemma A.10. The **MOVE-GROUP** rule operates strictly within the scope of the **moveGroup** operation.

Preserving well-formedness Let `oldParentID` be the ID of the group's parent feature in the original plan, and let

$$\begin{aligned}\text{FEATURES}'\,[\texttt{oldParentID}] &= (OP_e,\, OP_n,\, OP_t,\, OP_p,\, OP_c)\\ \text{FEATURES}'\,[\texttt{newParentID}] &= (NP_e,\, NP_n,\, NP_t,\, NP_p,\, NP_c)\\ \text{GROUPS}'\,[\texttt{groupID}] &= \left(G'_e,\, G'_t,\, G'_p,\, G'_c\right)\end{aligned}$$

Since the **MOVE-GROUP** rule does not remove or change the type of a feature, ***IBFM1*** and ***IBFM2*** hold. The modification made to the FEATURES map is

$$\begin{aligned}&\big(\text{FEATURES}\,[\texttt{oldParentID}]\\ &\leftarrow \texttt{removeGroupAt}\big(\text{FEATURES}\,[\texttt{oldParentID}]\,, [t_n, t_{p_2}), \texttt{groupID}\big)\big)\,[\texttt{newParentID}]\\ &\leftarrow \texttt{addChildGroup}\,\big(\text{FEATURES}\,[\texttt{newParentID}]\,, \texttt{groupID}, t_n\big)\end{aligned}$$

This change modifies only the subgroup maps of the original and new parent features of the group. In the modified model, for any time point t_i in the temporal scope, $\texttt{groupID} \notin OP_c\,[t_i]$, and $\texttt{groupID} \in NP\,[t_i]$. Furthermore, the GROUPS map is changed by

$$\begin{aligned}\text{GROUPS}\,[\texttt{groupID}] \leftarrow (&G_e,\, G_n,\, G_t,\\ &\texttt{clampInterval}(G_p, t_n)\,\big[[t_n, t_{p_2})\big] \leftarrow \texttt{newParentID},\, G_c)\end{aligned}$$

meaning that $G'_p\,[t_i] = \{\texttt{newParentID}\}$. Hence ***IBFM3*** and ***IBFM6*** hold.

As the group's types and subfeatures map are not modified, ***IBFM4–5*** and ***IBFM7*** are true for the modified model. Similarly, since the rule does not alter the group's existence set, ***IBFM8*** is preserved.

The intention of the cycle detection algorithm in Section 4.5.1 on page 39 is to uphold ***IBFM9***. Given the assumption that the original interval-based feature model contains no cycles, if the altered model contains a cycle then the **moveGroup** operation introduced it, and the group being moved must be part of the cycle. This could only happen if the group became part of its own subtree during the temporal scope, which means that at some point, the group occurs in its own list of ancestors. The algorithm looks at the group's *new* ancestors, meaning the ancestors that the group does not have in the original plan, but does in the new one. It then checks that none of those ancestors are moved to the group's subtree. Thus the rule preserves ***IBFM9***.

Lemma A.11. The **MOVE-GROUP** rule preserves well-formedness of the interval-based feature model.

Correctness of model modification The operation is intended to move the group with ID `groupID` to the feature with ID `newParentID` during the temporal scope $[t_n, t_{p_2})$. After applying the **MOVE-GROUP** rule, the only differences between the original and modified interval-based feature model should be

(i) The group's parent feature should be `newParentID` during the temporal scope

(ii) The group should not appear in the original parent feature's set of subgroups during the temporal scope

(iii) The group should appear in the new parent feature's set of subgroups

Given the modified map of parent features G'_p and the original map G_p, we have that

$$G'_p = \texttt{clampInterval}(G_p, t_n)\ [[t_n, t_{p_2})] \leftarrow \texttt{newParentID}$$

This statement assigns `newParentID` to the temporal scope $[t_n, t_{p_2})$ after applying $\texttt{clampInterval}(G_p, t_n)$, meaning that the original parent mapping is shortened to end at t_n, and a new mapping $[\,[t_n, t_{p_2}) \mapsto \texttt{newParentID}\,]$ is inserted. By semantics of assignment, it is clear that for t_i with $t_n \leq t_i < t_{p_2}$, $G'_p\,[t_i] = \{\texttt{newParentID}\}$, which is the desired result and fulfils (i).

By Lemma A.11 on the previous page and ***IBFM5***, (ii) and (iii) follow from (i). In other words, since the updated interval-based feature model is well-formed, and the group's parent feature during the temporal scope is `newParentID`, the group is not in the original parent feature's set of subgroups during the temporal scope, and is in the new parent feature's set of subgroups.

Lemma A.12. The **MOVE-GROUP** rule updates the interval-based feature model according to the semantics of the **moveGroup** operation.

A.5 Soundness of the Change Feature Variation Type Rule

See Figure 4.20 on page 42 for the **CHANGE-FEATURE-VARIATION-TYPE** rule. Let

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

be the initial state, and

$$(\text{NAMES}', \text{FEATURES}', \text{GROUPS}')$$

be the result state after applying the **CHANGE-FEATURE-VARIATION-TYPE** rule. Recall that this operation changes the type of the feature with ID `featureID` to `type`.

Scope Recall that the temporal scope of the **CHANGE-FEATURE-VARIATION-TYPE** rule is $[t_n, t_k)$ (Section 3.2 on page 28), where t_k is the time point at which the type is originally planned to be changed or the feature is removed. In the rule, this scope is identified by

$$F_t\,[t_n]_{\leq} = \{[t_{t_1}, t_{t_2})\}$$

Here, the time point t_n for changing the feature type is looked up in the feature's types map's set of interval keys, and the expected result is $\{[t_{t_1}, t_{t_2})\}$. This means that there is a mapping $[\,[t_{t_1}, t_{t_2}) \mapsto \texttt{oldType}\,]$ in F_t, with `oldType` being the type of the feature at time t_n, and this stops being the case at t_{t_2}. Thus the temporal scope of this operation is $[t_n, t_{t_2})$. The only interval looked up or assigned to in the rule is $[t_n, t_{t_2})$, so the rule operates strictly within the temporal scope of the operation.

The spatial scope for this operation is the feature itself and its parent group. Since the feature may move during the temporal scope, there may be several parent groups to consider. These groups and their types are looked up in the premise

$$\begin{aligned} &\forall [t_{p_1}, t_{p_2}) \in F_p\,[[t_n, t_{t_2})]_{\lessgtr} \\ &\forall p \in F_p\,\left[[t_{p_1}, t_{p_2})\right] \\ \forall t \in \texttt{getTypes}\left(\text{GROUPS}\,[p]\,, \langle [t_{p_1}, t_{p_2})\rangle_{t_n}^{t_{t_2}}\right)& \\ \left(\texttt{compatibleTypes}(t, \texttt{type})\right)& \end{aligned}$$

Otherwise, the only feature or group looked up or assigned to in the rule is $features\,[\texttt{featureID}]$, so the rule stays within the spatial scope.

Lemma A.13. The **CHANGE-FEATURE-VARIATION-TYPE** rule operates strictly within the scope of the **changeFeatureVariationType** operation.

Preserving well-formedness Due to the premise $\texttt{featureID} \neq \texttt{rootID}$, the feature is not the root, so ***IBFM1–2*** hold trivially. The modification to F_t

$$\texttt{clampInterval}(F_t, t_n)\,[[t_n, t_{t_2})] \leftarrow \texttt{type}$$

ensures that the feature's original type stops at t_n and the new one lasts for the duration of the temporal scope $[t_n, t_{t_2})$. Since the feature has exactly one type during the temporal scope, and no other modifications are made

to the feature, ***IBFM3*** is preserved. Because of this, and since the GROUPS map is also left unchanged, ***IBFM4–6*** and ***IBFM8–9*** hold.

As discussed in the **Scope** paragraph, the premise

$$\begin{aligned} &\forall [t_{p_1}, t_{p_2}) \in F_p\,[[t_n, t_{t_2})]_{\gtrless} \\ &\forall p \in F_p\,\left[[t_{p_1}, t_{p_2})\right] \\ \forall t \in \texttt{getTypes}\left(\text{GROUPS}\,[p]\,, \left\langle [t_{p_1}, t_{p_2}) \right\rangle_{t_n}^{t_{t_2}}\right) & \\ \left(\texttt{compatibleTypes}(t, \texttt{type})\right) & \end{aligned}$$

looks up all parent mappings overlapping the temporal scope ($[\,[t_{p_1}, t_{p_2}) \mapsto p\,]$), finds the types each parent group has during the scope and *while* it is the parent of the feature, and verifies that those types are compatible. Thus ***IBFM7*** is preserved.

Lemma A.14. The **CHANGE-FEATURE-VARIATION-TYPE** rule preserves well-formedness of the interval-based feature model.

Correctness of model modification The expected result of applying the rule is that $\text{FEATURES}'\,[\texttt{featureID}] = \left(F'_e, F'_n, F'_t, F'_p, F'_c\right)$ has the type `type` during the temporal scope $[t_n, t_{t_2})$. Indeed, due to the semantics of `clampInterval` and assignment, for any time point t_i such that $t_n \leq t_i < t_{t_2}$,

$$F'_t\,[t_i] = \{\texttt{type}\}$$

Since no other part of the interval-based feature model is altered, the rule performs as desired.

Lemma A.15. The **CHANGE-FEATURE-VARIATION-TYPE** rule updates the interval-based feature model according to the semantics of the **changeFeatureVariationType** operation.

A.6 Soundness of the Change Group Variation Type Rule

See Figure 4.22 on page 43 for the **CHANGE-GROUP-VARIATION-TYPE** rule. Let

$$\textbf{changeGroupVariationType}(\texttt{groupID}, \texttt{type}) \text{ at } t_n \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$

be the initial state, and

$$(\text{NAMES}', \text{FEATURES}', \text{GROUPS}')$$

be the result state after applying the **CHANGE-GROUP-VARIATION-TYPE** rule. Recall that this operation changes the type of the group with ID `groupID` to `type`.

Scope Recall that the temporal scope of the **CHANGE-GROUP-VARIATION-TYPE** rule is $[t_n, t_k)$ (Section 3.2 on page 28), where t_k is the time point at which the type is originally planned to be changed or the group is removed. In the rule, this scope is identified by

$$G_t\,[t_n]_{\leq} = \{[t_{t_1}, t_{t_2})\}$$

Here, the time point t_n for changing the group type is looked up in the group's types map's set of interval keys, and the expected result is $\{[t_{t_1}, t_{t_2})\}$. This means that there is a mapping $[\,[t_{t_1}, t_{t_2}) \mapsto \texttt{oldType}\,]$ in G_t, with `oldType` being the type of the group at time t_n, and this stops being the case at t_{t_2}. Thus the temporal scope of this operation is $[t_n, t_{t_2})$. The only interval looked up or assigned to in the rule is $[t_n, t_{t_2})$, so the rule operates strictly within the temporal scope of the operation.

The spatial scope for this operation is the group itself and its parent feature. The group may have several subfeatures during the temporal scope, which may both move and change their types. These features and their types are looked up in the premise

$$\begin{aligned} &\forall [t_{c_1}, t_{c_2}) \in G_c\,[[t_n, t_{t_2})]_{\lessapprox} \\ &\forall c \in \bigcup G_c\,[[t_{c_1}, t_{c_2})] \\ \forall t \in \texttt{getTypes}\,&\left(\text{FEATURES}\,[c]\,, \langle [t_{c_1}, t_{c_2})\rangle_{t_n}^{t_{t_2}}\right) \\ &\left(\texttt{compatibleTypes}(\texttt{type}, t)\right) \end{aligned}$$

Otherwise, the only feature or group looked up or assigned to in the rule is *groups* [`groupID`], so the rule stays within the spatial scope.

Lemma A.16. The **CHANGE-GROUP-VARIATION-TYPE** rule operates strictly within the scope of the **changeGroupVariationType** operation.

Preserving well-formedness The modification to G_t

$$\texttt{clampInterval}(G_t, t_n)\,[[t_n, t_{t_2})] \leftarrow \texttt{type}$$

ensures that the group's original type stops at t_n and the new one lasts for the duration of the temporal scope $[t_n, t_{t_2})$. Since the group has exactly one type during the temporal scope, ***IBFM4*** holds.

As discussed in the **Scope** paragraph, the premise

$$\begin{aligned} &\forall [t_{c_1}, t_{c_2}) \in G_c\,[[t_n, t_{t_2})]_{\gtrless} \\ &\forall c \in \bigcup G_c\,[[t_{c_1}, t_{c_2})] \\ \forall t \in \texttt{getTypes}\left(\text{FEATURES}\,[c]\,, \langle [t_{c_1}, t_{c_2})\rangle_{t_n}^{t_{t_2}}\right)& \\ \left(\texttt{compatibleTypes}(\texttt{type}, t)\right)& \end{aligned}$$

looks up all subfeature mappings overlapping the temporal scope ($[\,[t_{c_1}, t_{c_2}) \mapsto \{f_1, f_2, \ldots\}\,]$), finds the types each subfeature has during the scope and *while* it is the subfeature of the group, and verifies that those types are compatible. Thus ***IBFM7*** is preserved. As no changes are made to any other part of the interval-based feature model, the other requirements ***IBFM1–3***, ***IBFM5–6***, and ***IBFM8–9*** hold trivially.

Lemma A.17. The **CHANGE-GROUP-VARIATION-TYPE** rule preserves well-formedness of the interval-based feature model.

Correctness of model modification The expected result of applying the rule is that $\text{GROUPS}'\,[\texttt{groupID}] = \left(G'_e, G'_t, G'_p, G'_c\right)$ has the type `type` during the temporal scope $[t_n, t_{t_2})$. Indeed, due to the semantics of `clampInterval` and assignment, for any time point t_i such that $t_n \leq t_i < t_{t_2}$,

$$G'_t\,[t_i] = \{\texttt{type}\}$$

Since no other part of the interval-based feature model is altered, the rule performs as desired.

Lemma A.18. The **CHANGE-GROUP-VARIATION-TYPE** rule updates the interval-based feature model according to the semantics of the **changeGroupVariationType** operation.

A.7 Soundness of the Change Feature Name Rule

See Figure 4.23 on page 43 for the **CHANGE-FEATURE-VARIATION-TYPE** rule. Let

$$\begin{aligned} &\textbf{changeFeatureName}(\texttt{featureID}, \texttt{name}) \text{ at } t_n \triangleright \\ &\qquad (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \end{aligned}$$

be the initial state, and

$$(\text{NAMES}', \text{FEATURES}', \text{GROUPS}')$$

be the result state after applying the **CHANGE-FEATURE-NAME** rule. Recall that this operation changes the name of the feature with ID `featureID` to `name`.

Scope Recall that the temporal scope of the **CHANGE-FEATURE-NAME** rule is $[t_n, t_k)$ (Section 3.2 on page 28), where t_k is the time point at which the name is originally planned to be changed or the feature is removed. In the rule, this scope is identified by

$$F_n\,[t_n]_{\leq} = \{[t_{n_1}, t_{n_2})\}$$

Here, the time point t_n for changing the name is looked up in the feature's names map's set of interval keys, and the expected result is $\{[t_{n_1}, t_{n_2})\}$. This means that there is a mapping $[\,[t_{n_1}, t_{n_2}) \mapsto \texttt{oldName}\,]$ in F_n, with `oldName` being the name of the feature at time t_n, and this stops being the case at t_{n_2}. Thus the temporal scope of this operation is $[t_n, t_{n_2})$. The only interval looked up or assigned to in the rule is $[t_n, t_{n_2})$, so the rule operates strictly within the temporal scope of the operation.

The spatial scope for this operation is the name, the feature, and its original name. The only feature looked up or assigned to is FEATURES [`featureID`], and the only names looked up or assigned to are `oldName` and `name`. The GROUPS map is not modified or looked up in by the rule. Clearly, the rule stays within the spatial scope.

Lemma A.19. The **CHANGE-FEATURE-NAME** rule operates strictly within the scope of the **changeFeatureName** operation.

Preserving well-formedness The rule does not modify any feature's existence set or type, so ***IBFM1–2*** holds. Since it does change a name, we must look at that modification to make sure that ***IBFM3*** is true for the altered model. A requirement for ***IBFM3*** is that a feature has *exactly* one name. The feature is altered thus:

$$\begin{aligned}((\text{NAMES}\,[\texttt{name}]\,[[t_n, t_{n_2})] \leftarrow \texttt{featureID})\,[\texttt{oldName}] \leftarrow \\ \texttt{clampInterval}(\text{NAMES}\,[\texttt{oldName}]\,, t_n),\end{aligned}$$

This ensures that the feature's original name stops at t_n and the new one lasts for the duration of the temporal scope $[t_n, t_{n_2})$, ensuring that the feature has *exactly* one name during the temporal scope. Moreover ***IBFM3***

requires that the name belongs to the same feature, and no other. This is fulfilled by

$$\left(\text{NAMES}\left[\texttt{oldName}\right] \leftarrow \texttt{clampInterval}\left(\text{NAMES}\left[\texttt{oldName}\right], t_n\right)\right)\left[\texttt{name}\right]\left[[t_n, t_{n_2})\right] \leftarrow \texttt{featureID}$$

Here, the interval containing t_n in $\text{NAMES}\left[\texttt{oldName}\right]$ is clamped to end at t_n, and the resulting map is assigned `featureID` at `name` during the temporal scope, so the new name belongs to only the feature. This fulfils ***IBFM3***. As no other part of the interval-based feature model is modified, ***IBFM4–9*** hold.

Lemma A.20. The **CHANGE-FEATURE-NAME** rule preserves well-formedness of the interval-based feature model.

Correctness of model modification The expected result of applying the rule is that $\text{FEATURES}'\left[\texttt{featureID}\right] = \left(F'_e, F'_n, F'_t, F'_p, F'_c\right)$ has the name `name` during the temporal scope $[t_n, t_{n_2})$. Indeed, due to the semantics of `clampInterval` and assignment, for any time point t_i such that $t_n \leq t_i < t_{n_2}$,

$$F'_n\left[t_i\right] = \{\texttt{name}\}$$

Additionally, we should have $\text{NAMES}'\left[\texttt{name}\right] = \texttt{featureID}$. This is shown in the previous paragraph on well-formedness. Since no other part of the interval-based feature model is altered, the rule performs as desired.

Lemma A.21. The **CHANGE-FEATURE-NAME** rule updates the interval-based feature model according to the semantics of the **changeFeature-Name** operation.