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

Modular Soundness Checking of Feature Model Evolution Plans

Ida Sandberg Motzfeldt

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 modelled 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 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 verification tool for modification of FMEPs. This may be used as basis for a productive SPL planning tool.

Acknowledgements

I would like to thank my supervisors, Ingrid and Crystal, for helping and motivating me through the process of writing this thesis. They are both highly academically gifted and genuinely nice people. It has been a pleasure to work with them, both on the thesis and the research project. I am grateful that we got to know each other on a personal level through the research trips.

I would also like to thank Adrian, Christoph, and Michael for their help and support, and for welcoming us to Germany and Braunschweig for our collaboration. The trips to Germany and our meetings in Oslo have some of the highlights during my studies, both academically and socially. I especially enjoyed the Christmas market in Braunschweig. I am very proud to have my name on a published paper alongside all these great and very clever people.

The Department of Informatics has been an excellent place to study. I have made valuable and lasting friendships here, and met some of my favourite people. Academically, I have thrived here both as a student and as a teaching assistant, largely due to the warm social environment at this department.

I am grateful to Lars for our morning coffees during the pandemic, and for helping me design my favourite symbol $\in_{\lesssim}$. His enthusiasm inspires me.

A special thanks goes to Eirik, who has been my greatest support and friend during our studies. He has taught me a lot over the course of our various academic collaborations, patiently helping and encouraging me whenever my motivation has faltered. Our late night study sessions have been some of the best times I have had during the five years we have studied together. I am grateful to have had such a wonderful friend to write my thesis with.

Finally, I wish to thank my parents, who have always encouraged me to pursue my interests and helped me achieve my goals.

Ida Sandberg Motzfeldt
Oslo, 2021

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 also making 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 she believes will take 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 new 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. The illustration shows two evolution plans. In the original plan, a feature A exists in time 1 and is removed at time 5. We modify the plan by adding a child feature B to A at time 3. This change causes a paradox at time 5, since feature B is left without a parent feature. In this case, it would be simple to detect this paradox by hand, but given a plan with hundreds of features and points in time, paradoxes may be harder to locate. Thus, there is a need for tooling

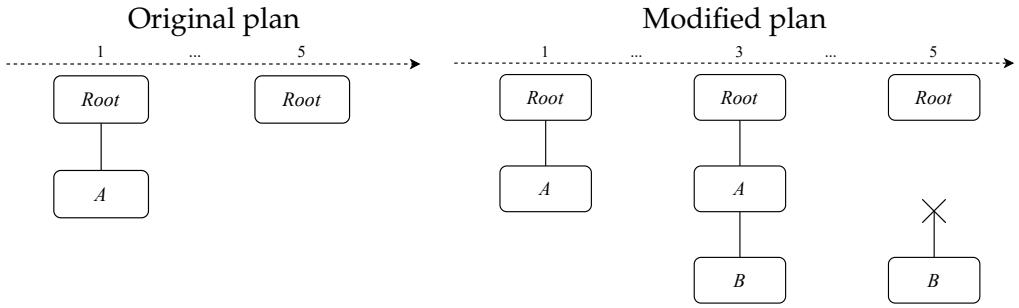


Figure 1.1: Simple paradox

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 the process of *evolving* the plan 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), as well as a framework for expressing and verifying logical relationships and dependencies between the spatial and temporal components of the plan.

This soundness verification 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 in [7], 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

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

3) Analyse the new plan in its entirety

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 much of the plan will often not be affected by a change. 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, in both dimensions. One is the *spatial* dimension, i.e., which parts of the feature model a change affects, and the other is the temporal dimension, i.e., which points in time in the plan are affected by change. We can then exploit that adding B only affects its parent parent feature A during the time between 3 and 5, ignoring the *Root* feature and time 1. The added benefit in this example is negligible, but for larger plans, ignoring hundreds of features and points in time will likely improve the performance significantly.

1.2 Research Questions

Although we have formalized the feature model evolution plan in our previous work [7], change to such a plan has not been addressed formally. The goal of this thesis is to formalize plan change and create an analysis method which verifies it, leveraging the knowledge that a change affects only parts of the plan. In order to achieve this goal, the thesis will address the following research questions.

RQ1 *Which operations are necessary 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 and formalize a feature model evolution plan in such a way that the scope of each operation can be captured?* 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* in space and time 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 *modularly*.

RQ3 *How can we soundly analyse change?* 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 update operations for changing feature model evolution plans. Furthermore, we 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 plans is devised with the aim to easily isolate the scope of an operation for analysis. Based on the scope and representations, we create an analysis method for validation and application of the update operations. The analysis is formalized as a set of rules, giving a detailed specification of when an operation may be applied to the evolution plan, and how to apply the modification. We implement a prototype of the analysis as proof of concept. Finally, we give a proof that the rule set is sound by showing that each rule preserves well-formedness of the structure of the feature model, that the application of each rule affects only a specified scope within the feature model evolution plan, and that each rule 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.

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.

Chapter 6 describes an implementation of the rules. We present the implementation by first giving an overview of the data types to provide intuition, and briefly present the translation of the analysis rules.

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

Chapter 2

Background

In this chapter, 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 example, all smartphones have technology for internet access, but other features are only included in some phones, like for instance a fingerprint reader. 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]

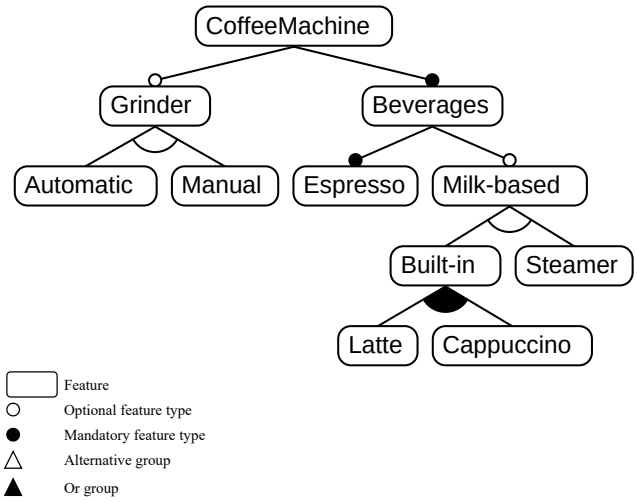


Figure 2.1: Example feature model for a coffee machine [1]

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 are also useful as documentation for an 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. A feature cannot be selected in a variant unless its parent feature is also selected [2]. See Figure 2.1 for an example of a feature model.

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 child feature 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

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

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 which are neither ALTERNATIVE nor OR, as for instance Beverages, have the type AND, which means that zero, one, or more of its child features 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, they are beyond the scope of this thesis.

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.

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

Furthermore, a feature model is a tree structure and must not contain cycles. Requirement ***WF8*** is not taken into account in this thesis. A

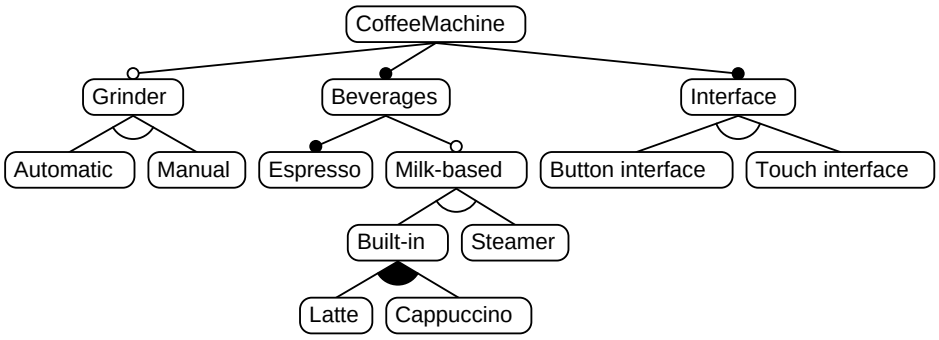


Figure 2.2: Coffee machine with added touch interface [2]

paradox is a violation of well-formedness requirements ***WF1–WF7***, and a plan without paradoxes is sound.

2.1.2 Feature Model Evolution Plans

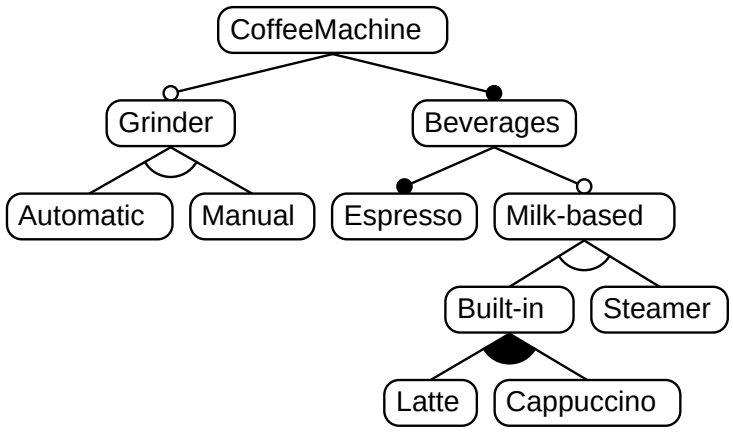
The evolution of an SPL can be planned using a *feature model evolution plan*. Software product lines often grow very large, and it is crucial to plan ahead. There exist tools for evolution planning, such as DarwinSPL [4]. An intuitive way to think of an evolution plan is as a sequence of feature models 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 10. 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 next page. The initial model associated with time t_0 is the one shown in Figure 2.1 on page 10. Applying the operations at t_1 results in the feature model shown in Figure 2.2. In the example, names are used in place of IDs to improve readability, but in the formal definitions, IDs are used to identify features and groups uniquely.

We have published a semantics for these edit operations along with a formal definition of a feature model, letting us formally define a sound

[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

plan [7]. 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 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, i.e., a plan resulting in no paradoxes. 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; 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]. As deciding properties about programs often reduce to the

famously undecidable halting problem, an algorithm cannot in general decide exactly how a program behaves [9]. However, there exist several methods to safely approximate solutions. For instance, live variable analysis discovers which variables *may* still be "alive" (meaning used in the future) at a certain point of the program, which may be used 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 cannot be overwritten safely. 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 are also *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, as in our publication [7], 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 iteration. The same goes for an operational feature model evolution plan. The plan has a finite number of steps and no iteration, and thus always will terminate. 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.

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–WF7*** given in Section 2.1.1 on page 11. 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 OR contain features of type MANDATORY, etc. This can be verified automatically, as we did in [7].

Part II

Definitions, Analysis, and Soundness Proofs

Chapter 3

Formalizing the Feature Model Evolution Plan

To achieve our goal of a modular analysis of modification for feature model evolution plans, we first need a representation that supports local lookup and modification. Using the representation we defined in the paper [7], with an initial model followed by a list of operations associated with time points, would not serve us, as the operations have to be applied in order to retrieve the state (current feature model) at any point in time. We present a representation for feature model evolution plans — the *interval-based feature model* — enabling lookup of information about specific parts of the feature models at specific times, as well as the data structures needed to define it. Furthermore, we formalise evolution plan change in terms of operations, and present the scope of each operation.

3.1 Interval-Based Feature Model

In this section we present the interval-based feature model as our representation for feature model evolution plans. To define it, we must first formally define the data structures it is based upon.

A feature model evolution plan has two dimensions: the spatial dimension and the temporal dimension. The spatial dimension consists of the feature models — which features and groups exist, what their names and types are, and how they are related. The temporal dimension concerns time, i.e., which points in time appear in the feature model evolution plan. To store the information about the spatial dimension, we have decided to use maps, which are useful for looking up information about a specific element. Looking up a feature ID in such a map will give us the information

about that feature.

Definition 3.1 (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 $[\, k \mapsto v \,] \in \text{MAP}$.

For example, in the map M from numbers to strings

$$\text{M} = \{[\, 1 \mapsto \text{“Static”} \,], [\, 2 \mapsto \text{“Analysis”} \,]\}$$

the keys are 1 and 2, and looking up the key 1 gives us the value “Static”. Using the map syntax, $\text{M}\,[1] = \text{“Static”}$.

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:

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

If we wish to replace the value at key 2 by “Electricity”, we have that

$$\text{M}\,[2] \leftarrow \text{“Electricity”} = \{[\, 1 \mapsto \text{“Static”} \,], [\, 2 \mapsto \text{“Electricity”} \,]\}$$

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. We use this operator to remove a specific value from a set at key k. 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. If $v \notin S$, then

$$\text{MAP} \setminus^{v} k = \text{MAP}$$

In other words, trying to remove a value which does not exist does not modify the map.

We define *time points* as the points in time used in a feature model evolution plan. A time point must be a member of a set $\mathcal{T}$ such that $<$ is a strict total order on $\mathcal{T}$. An example of such a set are the integers $\mathbb{Z}$, since $<$ on integers is a strict total order. Time points can also be dates or strings, as long as any set of time points can be ordered uniquely. In this thesis we use natural numbers for their simplicity, but in practice, a time point will usually be a date.

We choose to express the temporal dimension of the feature model evolution plan using *intervals*. An interval denotes a range in time, for instance, from Monday to Friday. This interval contains Tuesday, but not Sunday.

Definition 3.2 (Interval)**.** We define an interval as a set of time points between a lower bound and an upper bound, where the lower and upper bounds are time points. 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 $t_n \neq \infty$, we have that $t_n < \infty$.

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 in both intervals. For instance, $[2, 4)$ overlaps $[3, \infty)$, since they both contain the time point 3. Any interval $[t_n, t_m)$ with $n \geq m$ is *empty*, meaning that it contains no time points.

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 $[\mathbf{max}(t_{start}, t_l), \mathbf{min}(t_{end}, t_r))$. For instance, $\langle [3, \infty) \rangle_2^5 = [3, 5)$, which is the overlap between $[3, \infty)$ and $[2, 5)$.

To link the spatial and temporal dimensions of the feature model evolution plan, we use interval maps, which let us express what is true for a feature model during an interval. For instance, we can use an interval map to express that a feature has the name "Grinder" from time 1 to 5.

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

To look up values, one can either give an interval or a time point as key. Both will return sets of values. Looking up an interval returns the set of values associated with keys overlapping the interval. 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 $t_1 < t_2 < \ldots < t_5$) and non-overlapping keys:

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

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 $\text{IM}\,[[t_n, t_m)]_{\lessgtr}$ 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, if $t_n \geq t_m$, then

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

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

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

An interval map can be used to formalize change. An interval mapping $[\,[0, 18) \mapsto \mathit{child}\,]$, in the context of human age, can signify that a person starts being a child at age 0, and stops being a child at age 18.

In addition to interval maps, we use *interval sets* to express the temporal dimension of a feature model evolution plan. Like the interval maps, they can be used to show when something is true or changes in a feature model, but where the change is implicit.

Definition 3.4 (Interval set)**.** An interval set is a set of intervals (Definition 3.2 on page 21).

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

Notice that if $[t_n, t_m) \in \text{IS}$ then also $[t_n, t_m) \in_{\leq} \text{IS}$, and $[t_n, t_m) \in_{\lessapprox} \text{IS}$. Thus $\in$ is the most restrictive, and $\in_{\lessapprox}$ the least restrictive. For instance, given the interval set

$$S = \{[1, 3), [6, \infty)\}$$

we have that $[100, 1000) \notin S$, but $[100, 1000) \in_{\leq} S$, since $[6, \infty)$ contains $[100, 1000)$. Likewise, $[2, 7) \notin_{\leq} S$, but $[2, 7) \in_{\lessapprox} S$, since $[2, 7)$ overlaps $[1, 3)$.

We also define $\in_{\leq}$ for time points t_n, so that $t_n \in_{\leq} \text{IS}$ if some interval $[t_i, t_j) \in \text{IS}$ with $t_i \leq t_n < t_j$. In our example set S, $1 \in_{\leq} S$ and $256 \in_{\leq} S$, but 3 and $4 \notin_{\leq} S$.

IS $[t_n]_{\leq}$ returns the subset of IS containing t_n. For instance $S\,[5]_{\leq} = \emptyset$, and $S\,[2]_{\leq} = \{[1, 3)\}$.

To describe an entire feature model evolution plan, we define the *interval-based feature model*. It consists of three maps: NAMES, FEATURES, and GROUPS. The NAMES map contains *all* of the names used in the feature model, and which features they belong to during which times. Similarly, the FEATURES and GROUPS maps rely on interval maps to store all of the information about features and groups throughout the plan, respectively. The information is retrieved by looking up a name, a feature ID, or a group ID, which promotes the modularity of plan change verification.

Definition 3.5 (Interval-based feature model)**.** An interval-based feature model (IBFM) is defined as a triple (NAMES, FEATURES, GROUPS) where NAMES is a map from names to interval maps with feature ID values, 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 mainly modularity. As previously mentioned, the goal of this thesis is to minimize which parts of the plan are checked for paradoxes, as a change rarely affects more than a small part of the plan. It would then be suboptimal to represent a plan as a sequence of trees associated with time points, or an initial model followed by a sequence of operations. 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 and

name are unique at all times.

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.

Each feature, group, and name should be readily available, so as to make sure that names and IDs are unique at all times. However, all of them can be modified. A name may be used by several features at different times. A group may be moved or removed and its type may be changed. All of these operations can be applied to a feature, and its name can be changed as well. Thus all of this information must be captured in the map entries; if we look up a name, we should find all its usages, and if we look up a feature or a group, *all* the information about its variations must be available. We therefore design the map entries with this in mind.

The NAMES map has entries of the form $[\,\texttt{name} \mapsto \text{IM}\,]$, where the interval map IM contains mappings on the form $[\,[t_{\text{start}}, t_{\text{end}}) \mapsto \texttt{featureID}\,]$, 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.

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

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

The root feature's ID is constant for a interval-based feature model. We assume that it has been computed and represent it by referring to $RootID$. This is to avoid cluttering the representation with information that never changes.

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

The GROUPS map has entries of the form $[\texttt{groupID} \mapsto \textit{group entry}]$. A group has a type, a parent feature, and zero or more child features. These can all be defined in terms of intervals and collected into a 4-tuple (G_e, G_t, G_p, G_c) similarly to the feature entries, 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 $(\emptyset, \emptyset, \emptyset, \emptyset)$.

3.1.1 Example — Application of 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:

$$F_p\left[t_5\right] = \{\texttt{parentGroupID}\}$$

This is exactly the same as how we previously used $G_t\left[t_2\right]$. 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

$$F_p\left[t_5\right] = \emptyset$$

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.

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

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

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

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

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

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:

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

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:

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

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

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

Figure 3.2: Small interval-based feature model

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.

3.1.2 Example — Interval-Based Feature Model

A small example of an interval-based feature model can be found in Figure 3.2. 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.

The same plan can be viewed in Figure 3.3, where the feature model at different stages of the plan are shown at time 1 and 5 respectively. It is clear from this example that the interval-based feature model is better suited for manipulating the structure than reading it.

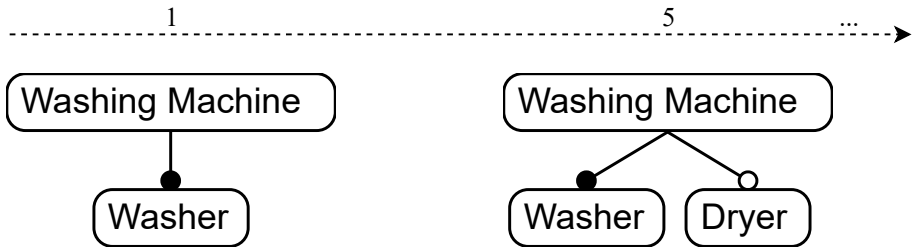


Figure 3.3: Washing machine visualisation

3.2 Operations

We define *update operations* to alter the interval-based feature model. The choice of operations is largely based on the edit operations defined in our earlier work [7]. We adapt them by adding a temporal dimension, letting us specify both *where* an operation should be applied in the feature model, and *when*, i.e. at which stage of the plan. We give a brief summary of the requirements a plan must fulfil for the operations to be applied.

- **addFeature**(`featureID`, `name`, `featureType`, `parentGroupID`) 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 $[t_n, t_m)$. No feature with ID `featureID` can exist during the interval, 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, i.e., if the feature has type MANDATORY, then the parent group must have type AND. We choose to let this operation affect the plan only within an interval so as to enable the adding of features to groups that are planned to be removed, and to add flexibility.

- **addGroup**(`groupID`, `groupType`, `parentFeatureID`) 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 cannot be in use during the interval, and the parent feature must exist during the entire interval.

- **removeFeature**(`featureID`) at t_n
  Removes the feature with ID `featureID` from the feature model at t_n. If the plan contains a removal of the feature and a subsequent reintroduction, removing the feature at an earlier stage does not affect the reintroduction, but rather moves the point of removal to an earlier point in time. The feature must exist at t_n in the original plan for the modification to be valid. The feature must not have any child groups that are left orphaned after removal.

- **removeGroup**(`groupID`) at t_n
  This operation is very similar to **removeFeature**. Removes the group with ID `groupID` from the feature model at t_n, not affecting potential later reintroductions. The group must exist at t_n in the original plan, and the group must not have any child features that are left orphaned after removal.

- **moveFeature**(`featureID`, `targetGroupID`) at t_n
  Moves the feature with ID `featureID` to the group with ID `targetGroupID` at t_n. The operation does not affect future moves planned for the feature. The feature's subtree is moved along with the feature. The move cannot be done if it introduces a cycle; that is, if the target group is in the feature's subtree at some point in the plan. Furthermore, the target group's type must be compatible with the feature's type, i.e. if the feature is MANDATORY and the group is OPTIONAL, the move cannot be done.

- **moveGroup**(`groupID`, `targetFeatureID`) at t_n
  This operation is very similar to **moveFeature**. It moves the group with ID `groupID` to the feature with ID `targetFeatureID` at t_n. The operation does not affect future moves planned for the group. The group's subtree is moved along with the group. If the move causes a

cycle, then the modification should not be applied.

- **changeFeatureVariationType**(`featureID`, `newType`) at t_n
  Changes the feature variation type of the feature with ID `featureID` to `newType` at time t_n. The change does not affect planned type changes to the feature. If the new type is MANDATORY, the parent group type must be AND, or else the operation cannot be applied.
- **changeGroupVariationType**(`groupID`, `newType`) at t_n
  Changes the group variation type of the group with ID `groupID` to `newType`. If the new type is OR or ALTERNATIVE, and a child feature has type MANDATORY, then the operation cannot be applied.
- **changeFeatureName**(`featureID`, `name`) at t_n
  Changes the name of the feature with ID `featureID` to `name`. It does not affect future renaming operations to the feature. No other feature may have the same name.

The operations given above cover most of the changes that are likely to be desired for a feature model evolution plan.

3.3 Temporal and Spatial Scopes of Update Operations

The scope of an operation consists of the parts of the plan that *may* be affected by the operation.

In this section, we define the scope for each of the operations defined in Section 3.2 on page 28. The scope is later used when creating the analysis rules for modification of the interval-based feature model. We describe the *spatial* and *temporal* scopes for each operation. The spatial scope consists of the parts of a feature model that may be affected by change. For instance, adding a group affects the group being added and its parent feature, as the operation does not modify any other part of the plan. We must also take into account the *temporal* aspect, since the plan also has a dimension of time. The temporal scope then consists of the time points that may be affected by applying an operation. In Figure 3.4[1] , we visualise the spatial and temporal scopes. In the original plan, a group D is added to feature B at time 2. In time 3, it is moved to feature C. We modify the plan by adding a feature E to group D from time 2 to time 4. In the modified plan, the spatial and temporal scopes are shown, with the spatial scope containing group D and feature E, and the temporal scope containing times 2 and 3.

[1]Notice that the plan differentiates between groups and features, but does not display types or names

Notice that *Root*, *A*, *B*, and *C* are not in the scope, nor are the time points 1 and 4.

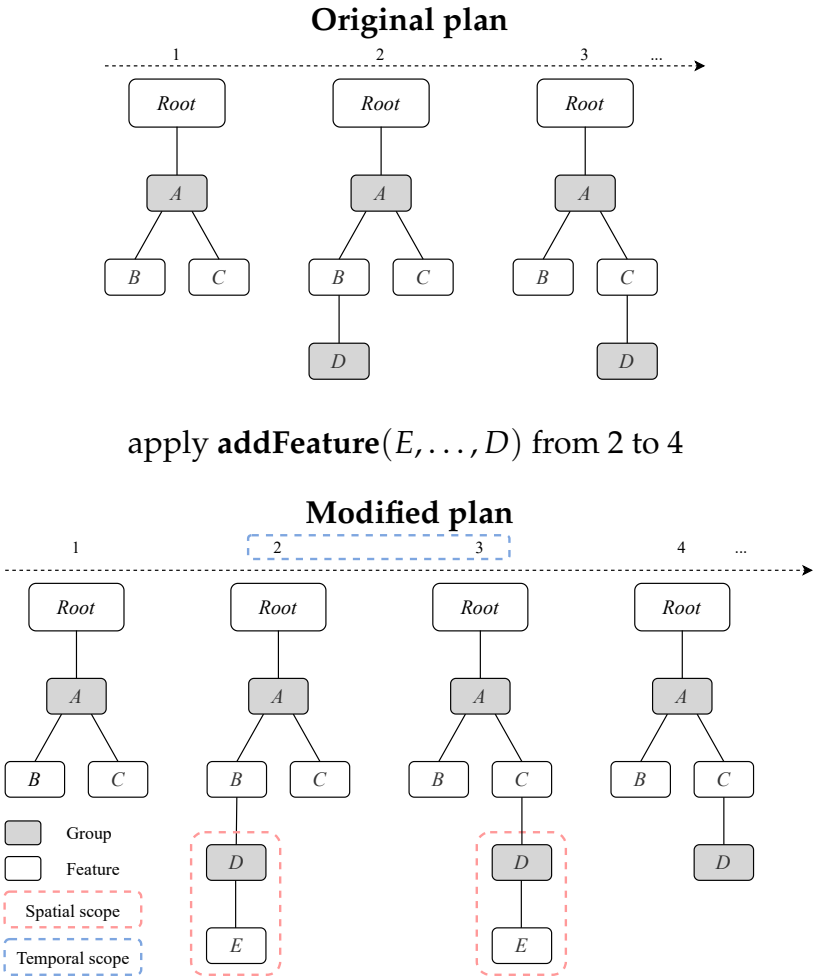


Figure 3.4: Add feature scope visualisation

For each of the operations defined in Section 3.2 on page 28, we define the scope in the temporal and spatial dimensions. By assuming that the original plan is sound, meaning that it contains no paradoxes, we include only those parts of the plan in which the operation may cause a paradox.

Operation Scopes

We define the temporal and spatial scopes for each operation.

- **addFeature**(`featureID`, `name`, `featureType`, `parentGroupID`) 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 plan is affected by the change. In other words, if

we look at the plan as a sequence of feature models, the only feature models that may become invalid 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`, `groupType`, `parentFeatureID`) 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
  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 $[t_n, t_k)$, where t_n is the time at which we specify that the feature be removed, and t_k is the time at which the feature was *originally* planned to be removed. In some cases, t_k will be ∞, meaning that the feature was never originally planned to be removed. 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, its child groups, and its name. If the feature has or will have a child group during the interval, then it cannot be removed. Otherwise, there are no conflicts. When modifying the interval-based feature model, the feature must be removed from the parent's set of child features, which is why the parent group is included in the spatial scope. Likewise, the feature's ID must be removed from its name's mappings during the temporal scope, and so the name is also inside the scope. If the name changes during the temporal scope, there is a paradox, since a feature planned to be modified should not be removed.

- **removeGroup**(`groupID`) at t_n
  This is similar to the scope for **removeFeature**, but without consideration for names, as groups do not have names. Thus the temporal scope is $[t_n, t_k)$, where t_k is the time at which the group was originally planned to be removed. The spatial scope includes the group itself, its parent feature, and its child features. As with features, a group cannot be removed if it has or will have child features during

the temporal scope.

- **moveFeature**(`featureID`, `targetGroupID`) at t_n
  If t_m is the time at which the feature is next moved or removed in the original plan, the temporal scope is $[t_n, t_m)$, since this operation only affects the plan within this interval. If the feature is *not* moved or removed in the original plan, then $t_m = \infty$.

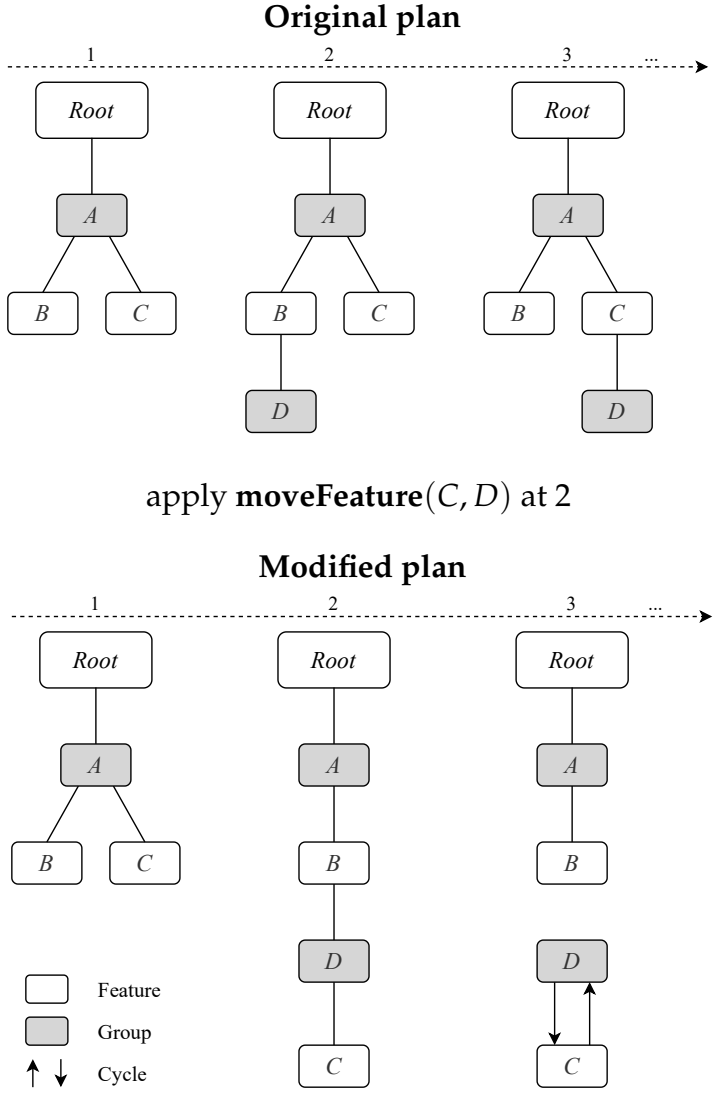


Figure 3.5: Move operation causing cycle

The spatial scope is discussed in more detail in the algorithm for moving features or groups presented in Section 4.5.1 on page 50. This scope is the largest and hardest to define, because we have to detect cycles. See Figure 3.5 for an example of a move which causes a cycle. In the example, the original plan contains a move operation in which group D is moved to feature C at time 3. In the modified

plan, feature C is moved to group D at time 2. Although this seems problem-free at time 2, it causes a paradox in the shape of a cycle at time 3, since D is then moved to a group in its subtree.

The scope is defined by the feature and its ancestors, as well as the target group and its ancestors, which may change during the intervals due to other move operations. It is not necessary to look at all ancestors, only the ones which `feature` and `targetGroup` do not have in common in the original plan, as well as the feature and the group themselves. Conflicting types and removal of the new parent must be considered in addition to cycles.

- **moveGroup**(`groupID`, `targetFeatureID`) at t_n
  This is similar to the scope for **moveFeature**, as cycles violate the tree structure independently of whether the nodes are features or groups. The difference in spatial scope concerns types, as type conflicts can only arise between a parent group and its child feature. Since the operation does not change the relation between a parent group and its child features, and the original plan is assumed to be sound, conflicting types are not considered for this operation.

- **changeFeatureVariationType**(`featureID`, `newType`) at t_n
  The temporal scope is $[t_n, t_m)$ if t_m is the next time point at which the feature's type changes next or when the feature is (next) removed, since this is the only part of the plan which is changed by the operation. Again, if no such change is planned, then the scope ends at ∞.

  The only possible conflict in the spatial scope is the parent group's type. At no point can the feature have type MANDATORY and the parent group have type ALTERNATIVE or OR. Thus, the spatial scope is the parent group and the feature itself.

- **changeGroupVariationType**(`groupID`, `newType`) at t_n
  The temporal scope is $[t_n, t_m)$ if t_m is the next time point at which the group's type changes next or when the group is (next) removed, since this is the only part of the plan which is changed by the operation.

  The spatial scope includes the group's child features; the possible conflict is the same as with changeFeatureType, but between the group and its child features. Consequently, this scope may encompass several features.

- **changeFeatureName**(`featureID`, `name`) at t_n
  The temporal scope is $[t_n, t_m)$ if t_m is the next time point at which the feature's name changes next or when the feature is (next) removed, since this is the only part of the plan which is changed by the

operation.

The spatial scope consists of 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 at time t_n, then the change is invalid.

When we present the analysis, the scope is used to decide which parts of the plan need to be examined. For each operation, we analyse and modify only the parts of the plan which are inside its scope.

Chapter 4

A Rule System for Analysis of Plan Change

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 have tool support that can check that the changes do not compromise the structure. Due to the size and complexity of the problem, it is impractical to let a human verify a change.

To communicate our analysis method, we use rules similar to structural operational semantics rules. The rules are on the form

(RULE-LABEL)

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

where S is the initial state, and S' is the new state after the rule is applied. The rule can only be applied if all the premises hold. In the rules, the initial 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. In all of the rules, we assume that the initial plan (NAMES, FEATURES, GROUPS) is sound, meaning that it contains no paradoxes. This lets us verify only the parts of the plan that are affected by the change, as the only paradoxes that can occur are the ones caused by the operation. The premises ensure that an operation can only be applied if some conditions hold; for instance the **ADD-FEATURE** rule in Figure 4.1 contains premises verifying that the feature does not already exist when we wish to add it. The rules give us

(ADD-FEATURE)

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

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

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

all the information we need to validate a modification, and to apply it.

4.1 Analysis Rule for Adding a Feature

Recall from Section 3.2 that the **addFeature** operation adds a feature to the feature model evolution plan during a given interval. Figure 4.1 describes the semantics of the **addFeature** operation.

When adding a feature during the interval $[t_n, t_m)$, its ID cannot be in use during the interval ($[t_n, t_m) \not\in_{\gtrless} F_e$). The parent group 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. When adding a feature, it may already exist in the plan during a different interval. In that case, we want to modify the existing feature entry. However, a feature being added is often completely new, and in that case it is useful that looking up the feature's ID in the FEATURES map returns an empty feature entry $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ instead of some undefined value, for instance $\perp$. This lets us treat both cases in the same way.

To make the rule tidier, we use three helper functions: `compatibleTypes` (in Figure 4.2), `setFeatureAttributes` (in Figure 4.3), and `addChildFeature`

(in Figure 4.4). The `compatibleTypes` function takes a group type (AND, OR or ALTERNATIVE) and a feature type (MANDATORY or OPTIONAL) and checks whether they are compatible. The types should belong to a parent group and its child feature. The only combination which is not allowed is a MANDATORY feature with an ALTERNATIVE or OR parent group.

The `setFeatureAttributes` function takes a feature entry, an interval, a name, a type, and a group ID, and returns the feature entry with the information included. It modifies the existence set by adding the interval, maps the interval to the name in the names map, to the type in the types map, and to the parent group ID in the parent groups map.

The `addChildFeature` function takes a group entry, an interval, and a feature ID, and adds the feature ID to the group's child feature map during the interval.

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

Figure 4.2: `compatibleTypes`

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

Figure 4.3: `setFeatureAttributes`

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

Figure 4.4: `addChildFeature`

4.1.1 Example — Application of the ADD-FEATURE Rule

We show how to use a rule by applying the **ADD-FEATURE** rule to a simple example. Following is a simple interval-based feature model, containing one feature and one group. The model is visualised in a simplified version in Figure 4.5. The visualisation shows only the structure of the feature model, not the feature or groups' types or other attributes.

$$
\begin{aligned}
&(\ \{[\,\text{Root} \mapsto \{[\,[1,\infty) \mapsto \text{feature:root}\,]\}\,] \\
&\ \ \} \\
&,\{[\text{feature:root} \mapsto \\
&\qquad\qquad (\ \{[1,\infty)\} \\
&\qquad\qquad ,\{[\,[1,\infty) \mapsto \text{Root}\,]\} \\
&\qquad\qquad ,\{[\,[1,\infty) \mapsto \textsc{mandatory}\,]\} \\
&\qquad\qquad ,\emptyset \\
&\qquad\qquad ,\{[\,[1,\infty) \mapsto \{\text{group:A}\}\,]\}\ )] \\
&\ \ \} \\
&,\{[\text{group:A} \mapsto \\
&\qquad\qquad (\ \{[1,5)\} \\
&\qquad\qquad ,\{[\,[1,5) \mapsto \textsc{and}\,]\} \\
&\qquad\qquad ,\{[\,[1,5) \mapsto \text{feature:Root}\,]\} \\
&\qquad\qquad ,\emptyset)] \\
&\ \ \})
\end{aligned}
$$

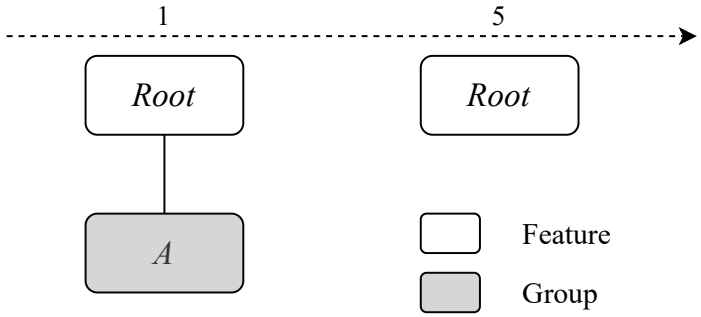


Figure 4.5: Add feature example — original plan

We apply the operation **addFeature**(feature:B, B, OPTIONAL, group:A) from 3 to 5 using the **ADD-FEATURE** rule. This means that we want to add a feature with ID feature:B, name B, and type OPTIONAL to the group with ID group:A from time 3 until (but not including) 5. We must go through all the premises and check that each of them holds for the original model. First, we look up the new feature in the original model. Since the feature does not exist, we get

$$\textsc{features}\,[\text{feature:B}] = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

We further look up the parent group:

$$\begin{aligned}\text{GROUPS}\,[\text{group:A}] = (\,&\{[1,5)\}\\ &,\,\{[\,[1,5) \mapsto \text{AND}\,]\}\\ &,\,\{[\,[1,5) \mapsto \text{feature:Root}\,]\}\\ &,\emptyset)\end{aligned}$$

We see that the group exists from 1 to 5, that it has the type AND during its lifespan, and that its parent feature has ID feature:Root. Now that we have the information we require, we can check that the other premises hold.

In the premise $[t_n, t_m) \not\in_{\lessgtr} F_e$, we replace t_n, t_m and F_e by their concrete values:

$$[3,5) \not\in_{\lessgtr} \emptyset$$

Since no element is member of $\emptyset$, no element in $\emptyset$ overlaps $[3,5)$, so this premise holds.

The premise $[t_n, t_m) \in_{\leq} G_e$ asserts that an interval in G_e should contain the interval $[t_n, t_m)$. We instantiate the variables with our concrete values and verify that this holds:

$$[3,5) \in_{\leq} \{[1,5)\}$$

Since $[1,5)$ does contain $[3,5)$, this premise also holds.

The premise $\text{NAMES}\,[\texttt{name}]\,[[t_n, t_m)] = \emptyset$ means that the name of the new feature should be unique during the temporal scope. We verify:

$$\text{NAMES}\,[\text{B}]\,[[3,5)] = \emptyset$$

The only key in the NAMES map is A, so this statement is true. The next premise we check is $t_n < t_m$, which is true because $3 < 5$.

Lastly, we check that the types of the new feature and its parent group are compatible, the premise $\forall \texttt{gt} \in G_t\,[[t_n, t_m)]\,(\texttt{compatibleTypes}(\texttt{gt}, \texttt{type}))$. We instantiate the variables:

$$\forall \texttt{gt} \in \{[\,[1,5) \mapsto \text{AND}\,]\}\,[[3,5)]\,(\texttt{compatibleTypes}(\texttt{gt}, \text{OPTIONAL}))$$

The only value in $\{[\,[1,5) \mapsto \text{AND}\,]\}$ during the interval $[3,5)$ is AND, so we instantiate gt with that value.

$$\texttt{compatibleTypes}(\text{AND}, \text{OPTIONAL})$$

This matches the first case in the compatibleTypes function (Figure 4.2), which evaluates to True. Thus the last premise holds, so we can apply the

transition. We go through each map individually, starting with NAMES. The rule gives the modification

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

which becomes

$$\{[\,\text{Root} \mapsto \{[\,[1,\infty) \mapsto \text{feature:root}\,]\}\,]\}\,[\text{B}]\,[[3,5)] \leftarrow \text{feature:B}$$

This gives us the map

$$\begin{aligned}\{[\,\text{Root} \mapsto \{[\,[1,\infty) \mapsto \text{feature:root}\,]\}\,],\\ [\,\text{B} \mapsto \{[\,[3,5) \mapsto \text{feature:B}\,]\}\,]\}\end{aligned}$$

According to the rule, the FEATURES map should be modified in the following way:

$$\begin{aligned}\text{FEATURES}\,[\texttt{featureID}] \leftarrow \texttt{setFeatureAttributes}(&\text{FEATURES}\,[\texttt{featureID}]\,,\\ &[t_n, t_m),\\ &\texttt{name}, \texttt{type}, \texttt{parentGroupID})\end{aligned}$$

We instantiate:

$$\begin{aligned}\text{FEATURES}\,[\text{feature:B}] \leftarrow \texttt{setFeatureAttributes}(&(\emptyset,\emptyset,\emptyset,\emptyset,\emptyset),\\ &[3,5),\\ &\text{A}, \text{OPTIONAL}, \text{group:A})\end{aligned}$$

We apply `setFeatureAttributes`:

$$\begin{aligned}&\texttt{setFeatureAttributes}((\emptyset,\emptyset,\emptyset,\emptyset,\emptyset), [3,5), \text{B}, \text{OPTIONAL}, \text{group:A})\\ &\quad = (\ \emptyset \cup \{[3,5)\}\\ &\qquad ,\ \emptyset\,[[3,5)] \leftarrow \text{B}\\ &\qquad ,\ \emptyset\,[[3,5)] \leftarrow \text{OPTIONAL}\\ &\qquad ,\ \emptyset\,[[3,5)] \leftarrow \text{group:A}\\ &\qquad ,\ \emptyset\ )\end{aligned}$$

This gives us the feature mapping

$$\begin{aligned}\text{FEATURES}\,[\text{feature:B}] \mapsto (&\ \{[3,5)\}\\ &,\{[\,[3,5) \mapsto \text{B}\,]\}\\ &,\{[\,[3,5) \mapsto \text{OPTIONAL}\,]\}\\ &,\{[\,[3,5) \mapsto \text{group:A}\,]\}\\ &,\emptyset)\end{aligned}$$

which is then mapped to FEATURES [feature:B].

The GROUPS map is modified by the rule in the following way:

$$\text{GROUPS}[\texttt{parentGroupID}] \leftarrow \texttt{addChildFeature}(\text{GROUPS}\,[\texttt{parentGroupID}]\,, [t_n, t_m), \texttt{featureID})$$

We substitute with our values:

$$\text{GROUPS}[\text{group:A}] \leftarrow \texttt{addChildFeature}((\{[1,5)\}\,, \{[\,[1,5) \mapsto \text{AND}\,]\}\,, \{[\,[1,5) \mapsto \text{feature:Root}\,]\}\,, \emptyset), [3,5), \text{feature:B})$$

We apply `addChildFeature`:

$$\begin{aligned}&\texttt{addChildFeature}((\{[1,5)\}\,, \{[\,[1,5) \mapsto \text{AND}\,]\}\,, \{[\,[1,5) \mapsto \text{feature:Root}\,]\}\,, \emptyset),\\ &\qquad [3,5),\ \text{feature:B})\\ =\ &(\{[1,5)\}\,, \{[\,[1,5) \mapsto \text{AND}\,]\}\,, \{[\,[1,5) \mapsto \text{feature:Root}\,]\}\,,\\ \hookrightarrow\ &\quad \{[\,[3,5) \mapsto \{\text{feature:B}\}\,]\})\end{aligned}$$

We then end up with the following interval-based feature model:

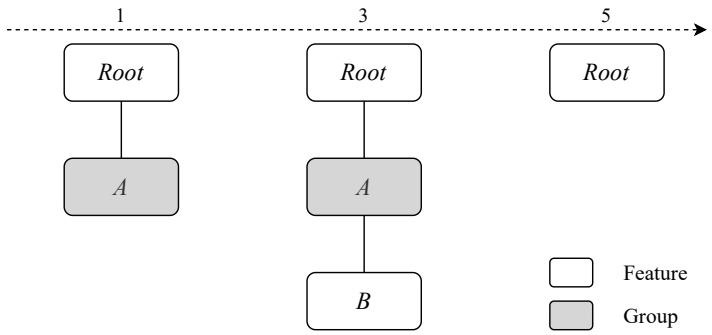


Figure 4.6: Add Feature — modified plan

$$
\begin{array}{l}
(\ \{[\,\text{Root} \mapsto \{[\,[1,\infty) \mapsto \text{feature:root}\,]\}\,] \\
\quad ,[\,\text{B} \mapsto \{[\,[3,5) \mapsto \text{feature:B}\,]\}\,] \\
\quad \} \\
\quad ,\{[\text{feature:root} \mapsto \\
\qquad\qquad\qquad (\ \{[1,\infty)\} \\
\qquad\qquad\qquad ,\{[\,[1,\infty) \mapsto \text{Root}\,]\} \\
\qquad\qquad\qquad ,\{[\,[1,\infty) \mapsto \textsc{mandatory}\,]\} \\
\qquad\qquad\qquad ,\emptyset \\
\qquad\qquad\qquad ,\{[\,[1,\infty) \mapsto \{\text{group:A}\}\,]\}\ )] \\
\\
\quad ,[\text{feature:B} \mapsto \\
\qquad\qquad\qquad (\ \{[3,5)\} \\
\qquad\qquad\qquad ,\{[\,[3,5) \mapsto \text{B}\,]\} \\
\qquad\qquad\qquad ,\{[\,[3,5) \mapsto \textsc{optional}\,]\} \\
\qquad\qquad\qquad ,\{[\,[3,5) \mapsto \text{group:A}\,]\} \\
\qquad\qquad\qquad ,\emptyset)] \\
\quad \} \\
\quad ,\{[\text{group:A} \mapsto \\
\qquad\qquad\qquad (\ \{[1,5)\} \\
\qquad\qquad\qquad ,\{[\,[1,5) \mapsto \textsc{and}\,]\} \\
\qquad\qquad\qquad ,\{[\,[1,5) \mapsto \text{feature:Root}\,]\} \\
\qquad\qquad\qquad ,\{[\,[3,5) \mapsto \{\text{feature:B}\}\,]\}\ )] \\
\quad \})
\end{array}
$$

This plan is visualised in Figure 4.6.

If we tried to add the feature from 3 to 6 instead, the premise $[t_n, t_m) \in_{\leq} F_e$ would fail, since the parent group only exists until 5. If a premise is false, the rule cannot be applied.

(ADD-GROUP)

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

Figure 4.7: The **ADD-GROUP** rule

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

Figure 4.8: `addChildGroup`

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

Figure 4.9: `setGroupAttributes`

4.2 Analysis Rule for Adding a Group

In Section 3.2 we have defined the **addGroup** operation, which adds a group to a feature model evolution plan during a given interval. The rule in Figure 4.7 describes the conditions which must be in place to add a group to the FMEP during an interval ($[t_n, t_m)$), as well as how to update the model if all the premises are true.

The group must not already exist in the plan during the interval ($[t_n, t_m) \not\in_{\trianglelefteq} G_e$), and the parent feature must exist for the duration of the interval ($[t_n, t_m) \in_{\leq} F_e$). Lastly, the interval must fulfil the condition $t_n < t_m$, meaning that it starts strictly before it ends.

If all the premises hold, the model is updated according to the semantics of the **addGroup** operation. The group ID is added to the parent feature's map of child groups with the interval as key, and the attributes specified in the operation are added to the group entry in the GROUPS map. This is achieved by using the `addChildGroup` function (in Figure 4.8), which is extremely similar to `addChildFeature`, and adds the group to its new parent feature, and the `setGroupAttributes` function (in Figure 4.9), which is similar to `setFeatureAttributes` and updates the group entry with the attributes given in the operation.

4.3 Analysis Rule for Removing a Feature

Recall that the **removeFeature** operation removes a feature at a given time, as defined in Section 3.2. Figure 4.10 shows the semantics of removing a feature with ID `featureID` at time t_n. We find the time point when the feature was to be removed in the original plan by looking up the interval containing t_n in the feature's EXISTENCE set $[t_{e_1}, t_{e_2})$. The interval in which the new plan is different from the original is then $[t_n, t_{e_2})$. We verify that the feature does not have any child groups during the affected interval ($F_c\,[[t_n, t_{e_2})] = \emptyset$). We furthermore check that the feature has only a single name, type, and parent during the interval. This means that the original plan did not change the feature's name, type, or parent during this time. If these conditions all hold, we update the interval-based feature model by clamping all the relevant intervals to t_n, i.e. shortening them to end at t_n. To achieve this, we use helper functions. The `clampInterval` function (in Figure 4.11) takes an interval map and a time point t_n, and shortens the key containing t_n to end at t_n. Note that it assumes that the interval map contains exactly one key containing the time point. The premises in the rule ensure that this is true. We use the `clampFeature` function (in Figure 4.14) to update the feature entry, ending all its interval map

(REMOVE-FEATURE)

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

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

Figure 4.10: The **REMOVE-FEATURE** rule

keys containing t_n at t_n by using `clampInterval` and `clampSetInterval` (in Figure 4.13). The helper function `clampSetInterval` does the same as `clampInterval`, but shortens a member of an interval set instead of a key in an interval map. To remove the feature from its parent group's child feature map, we use the helper function `removeFeatureAt` (in Figure 4.16). This function applies `clampIntervalValue` (in Figure 4.12) to the group's child feature map. The `clampIntervalValue` function removes the feature from the mapping containing the time point given, and adds it to the set at the key which ends at the same time point.

4.4 Analysis Rule for Removing a Group

We defined the operation **removeGroup** in Section 3.2. The operation removes a group at a given time point, similarly to **removeFeature**.

The **REMOVE-GROUP** rule in Figure 4.18 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 should not have any children, and there cannot be plans to change the type or move the group within the interval. We check the latter by looking up the type and parent feature during the interval; if the set contains only one type/parent feature then the type and parent feature do not change.

We use the helper functions `removeGroupAt` (in Figure 4.17) and `clampGroup` (in Figure 4.15) to update the interval-based feature model. The `removeGroupAt` function, similar to `removeFeatureAt`, takes the parent feature entry, the group ID, and a time point t_c, and applies

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

Figure 4.11: `clampInterval`

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

Figure 4.12: `clampIntervalValue`

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

Figure 4.13: `clampSetInterval`

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

Figure 4.14: `clampFeature`

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

Figure 4.15: `clampGroup`

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

Figure 4.16: `removeFeatureAt`

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

Figure 4.17: `removeGroupAt`

(REMOVE-GROUP)

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

$$\textbf{removeGroup}\,(\texttt{groupID})\ \text{at}\ t_n \triangleright$$
$$(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$
$$\longrightarrow$$
$$(\text{NAMES},$$
$$\text{FEATURES}\,[\texttt{parentFeatureID}] \leftarrow \texttt{removeGroupAt}\,(\text{FEATURES}\,[\texttt{parentFeatureID}]\,, \texttt{groupID}, t_n)\,,$$
$$\text{GROUPS}\,[\texttt{groupID}] \leftarrow \texttt{clampGroup}\,(\text{GROUPS}\,[\texttt{groupID}]\,, t_n)\,)$$

Figure 4.18: The **REMOVE-GROUP** rule

`clampIntervalValue` to the parent feature's child group map. The `clampGroup` function does the same as `clampFeature`, but to a group entry. After applying these to the interval-based feature model, the group is removed from the model during the temporal scope.

4.5 Analysis Rule for Moving a Feature

In Section 3.2, we defined the **moveFeature** operation to move a feature from a group to a new group at a given time. See Figure 4.19 for the semantics of the **moveFeature** operation. The premise $\neg\texttt{createsCycle}$ refers to the cycle detection algorithm described in Section 4.5.1 on the following page. A concrete implementation of the algorithm can be found on GitHub[1].

The premise $F_p\,[t_n]_{\leq} = \{[t_{p_1}, t_{p_2})\}$ locates the scope of the operation, namely $[t_n, t_{p_2})$. The ID of the feature's former parent group is identified in the premise $F_p\,[[t_n, t_{p_2})] = \{\texttt{oldParentID}\}$ for the purpose of updating the GROUPS map. The premise $[t_n, t_{p_2}) \in_{\leq} G_e$ ensures that the new parent exists during the entire temporal scope.

As the plan may contain several type changes for both the feature being moved and its new parent, we must check that the types they have at the same time are compatible. This is achieved by the following premise:

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

[1] `https://github.com/idamotz/Master/blob/master/soundness-checker/`

(MOVE-FEATURE)

$$\frac{\begin{array}{c}\neg\texttt{createsCycle} \qquad F_p\,[t_n]_{\leq} = \{[t_{p_1}, t_{p_2})\} \\ F_p\,[[t_n, t_{p_2})] = \{\texttt{oldParentID}\} \qquad [t_n, t_{p_2}) \in_{\leq} G_e \\ \\ \forall [t_{f_1}, t_{f_2}) \in F_t\,[[t_n, t_{p_2})]_{\lessapprox}\; \forall [t_{g_1}, t_{g_2}) \in G_t \left[\left\langle [t_{f_1}, t_{f_2}) \right\rangle_{t_n}^{t_{p_2}}\right]_{\lessapprox} \\ \forall \texttt{ft} \in F_t\left[[t_{f_1}, t_{f_2})\right]\, \forall \texttt{gt} \in G_t\,[[t_{g_1}, t_{g_2})]\,(\texttt{compatibleTypes}(\texttt{gt}, \texttt{ft})) \\ \\ \text{FEATURES}\,[\texttt{featureID}] = (F_e, F_n, F_t, F_p, F_c) \\ \text{GROUPS}\,[\texttt{newParentID}] = (G_e, G_t, G_p, G_c)\end{array}}{\begin{array}{c}\textbf{moveFeature}\,(\texttt{featureID}, \texttt{newParentID})\ \text{at}\ t_n \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \\ \longrightarrow \\ (\text{NAMES}, \\ \text{FEATURES}\,[\texttt{featureID}] \leftarrow (F_e, F_n, F_t, \texttt{clampInterval}(F_p, t_n)\,[[t_n, t_{p_2})] \leftarrow \texttt{newParentID}, F_c)\,, \\ (\text{GROUPS}\,[\texttt{oldParentID}] \\ \leftarrow \texttt{removeFeatureAt}\,(\text{GROUPS}\,[\texttt{oldParentID}], \texttt{featureID}, t_n))\,[\texttt{newParentID}] \\ \leftarrow \texttt{addChildFeature}(\text{GROUPS}\,[\texttt{newParentID}], [t_n, t_{p_2}), \texttt{featureID})\end{array}}$$

Figure 4.19: The **MOVE-FEATURE** rule

It says that for each interval key overlapping the temporal scope in the feature's type map, then for each interval in the group's type map overlapping the aforementioned key and restricted by the temporal scope, then for all types mapped to by those keys, those types must be compatible. This ensures that the rule is not too strict, because it check only those combinations of types which the feature and its new parent group have at the same time, further restricted by the temporal scope.

If all the premises hold, then the interval-based feature model is updated to reflect it. The feature's parent group map is updated by shortening the interval mapped to the former parent's ID to end at t_n, and adding a new mapping $[\,[t_n, t_{p_2}) \mapsto \texttt{newParentID}\,]$.

The feature is removed from the previous parent's (`oldParentID`) set of child features during the temporal scope, and the feature is added to the new parent's set of child features during the same interval.

4.5.1 Algorithm for Detecting Cycles Resulting from Move Operations

Compared with the other operations, **moveFeature** and **moveGroup** require extensive verification, as moving a feature or a group may cause

cycles at the time of the move or at some later point. Since cycles violate only the tree structure of the feature model evolution plan, we abstract away from groups and features, viewing both as nodes.

In Figure 4.20, we show an example of a cycle arising from a move operation. In the original plan, the marked node E is moved to C at time 2, and to *some node* in node D's subtree s. In the modified plan, D (and its subtree s) has been moved to node F. There is no cycle at time 1 or 2, but at time 3, when E is moved to s, it causes a cycle. At time 1, the new ancestors of D are F, G, and E. At time 2, the new ancestors are still F, G, and E, but C is added to the list, since C was not an ancestor of D in the original plan. When E is then moved to D's subtree s, it causes a cycle, since E now has F as an ancestor.

Following is a description of an algorithm intended to ensure that adding a **moveFeature** or **moveGroup** operation does not cause a cycle.

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 when n is moved next or removed, or ∞. We use the function $\texttt{ancestors}(IBFM, \texttt{node}, \texttt{time})$ (see Figure 4.21), written in Haskell-like syntax, which takes the interval-based feature model (IBFM), a node, and a time point and returns a list of `node`'s ancestors at time point t_n. It does this by following the parent references upwards at the time point t_n, alternating between features and groups. It stops when it reaches the root feature, which has no parent node.

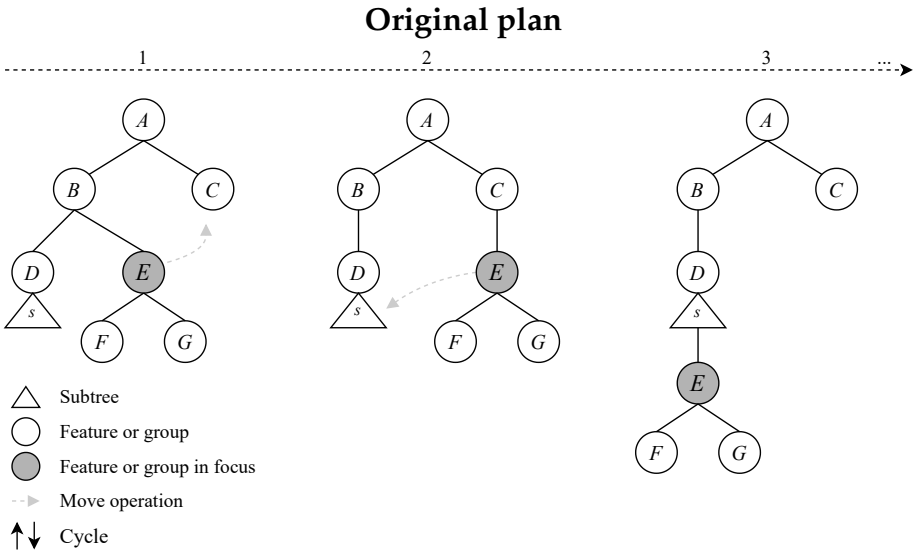
First, check whether $n \in \texttt{ancestors}(IBFM, c_1, t_1)$. If this is the case, report that the move causes a cycle and terminate.

Next, find a list of critical nodes. These are the nodes which may cause a cycle if they are moved. 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, following the order of the list C. If no such moves occur until t_e, the operation creates no cycles, and the algorithm terminates successfully. Suppose there is a move operation scheduled for t_k, with $t_1 \leq t_k < t_e$, 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 will cause a cycle and terminate.
2. k is not in n's subtree, so this move is safe. Let $A_k =$



apply **moveFeature**(D, F) at 1

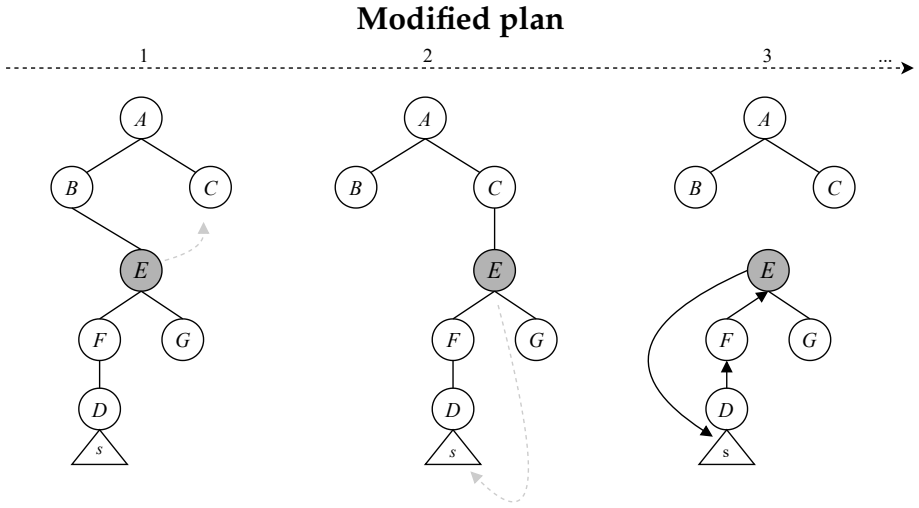


Figure 4.20: Illustration of move paradox

$\texttt{ancestors}(IBFM, k, t_k) = [k_1, k_2, \ldots, k_n, SN', \ldots, r]$, with SN′ the first common element of A_k and A_n. Update the list of critical nodes to $[c_1, \ldots, c_i, k_1, \ldots, k_n]$.

$$
\begin{array}{l}
\texttt{ancestors}\big((\text{NAMES}, \text{FEATURES}, \text{GROUPS}), \texttt{featureID}, t_n\big) \\
\quad = \texttt{case parentGroup of} \\
\qquad \texttt{\{ parentGroupID \}} \rightarrow \\
\qquad\quad \texttt{parentGroupID} : \texttt{ancestors}\big((\text{NAMES}, \text{FEATURES}, \text{GROUPS}), \\
\qquad\qquad\qquad\qquad\qquad \texttt{parentGroupID}, t_n\big) \\
\qquad \emptyset \rightarrow \texttt{[]} \\
\quad \texttt{where } \big(F_e, F_n, F_t, F_p, F_c\big) = \text{FEATURES}\,[\texttt{featureID}] \\
\qquad \texttt{parentGroup} = F_p\,[t_n]
\end{array}
$$

$$
\begin{array}{l}
\texttt{ancestors}\big((\text{NAMES}, \text{FEATURES}, \text{GROUPS}), \texttt{groupID}, t_n\big) \\
\quad = \texttt{parentFeatureID} : \texttt{ancestors}\big((\text{NAMES}, \text{FEATURES}, \text{GROUPS}), \\
\qquad\qquad\qquad\qquad\qquad \texttt{parentFeatureID}, t_n\big) \\
\quad \texttt{where } \big(G_e, G_t, G_p, G_c\big) = \text{GROUPS}\,[\texttt{groupID}] \\
\qquad \texttt{\{ parentFeatureID \}} = G_p\,[t_n]
\end{array}
$$

Figure 4.21: `ancestors`

(MOVE-GROUP)

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

Figure 4.22: The **MOVE-GROUP** rule

4.6 Analysis Rule for Moving a Group

The **moveGroup** operation moves a group from its parent feature to a new parent, as explained in Section 3.2.

See Figure 4.22 for the semantics of the **moveGroup** operation. The rule is similar to the **MOVE-FEATURE** rule, but it differs in that it does not have a check for types. This is because there can only be a type 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. The model is updated similarly to the way it is done in the **MOVE-FEATURE** rule. Here as well, the premise $\neg$`createsCycle` refers to the algorithm in Section 4.5.1. The reason this algorithm can be applied to both features and groups is that cycles break the *tree structure* of the feature model, which consists of both feature and group nodes. Hence, when checking for cycles, features and groups are treated the same.

(CHANGE-FEATURE-VARIATION-TYPE)

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

Figure 4.23: The **CHANGE-FEATURE-VARIATION-TYPE** rule

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

Figure 4.24: `getTypes`

4.7 Analysis Rule for Changing the Variation Type of a Feature

The operation **changeFeatureVariationType** is defined in Section 3.2 to update a feature's type at a given time point in the evolution plan.

The rule in Figure 4.23 shows the semantics of changing the feature variation type of the feature with ID `featureID` at time t_n. The first premise `featureID` $\neq$ *RootID* ensures that we are not attempting to modify the type of the root feature, which should always be MANDATORY. The second premise ($F_t\,[t_n]_{\leq} = \{[t_{t_1}, t_{t_2})\}$) identifies the upper bound of the temporal scope, t_{t_2}. This is when the feature type was originally planned to change. If there is no such planned change, then $t_{t_2} = \infty$. The temporal scope is then $[t_n, t_{t_2})$ as defined in Section 3.3. The next premise is a little convoluted, but its intent is easier to understand:

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

It checks that all the types a parent group has *while it is the parent of the feature*, restricted by the temporal scope, has a type which is compatible with the new type of the feature. This is necessary as the feature may potentially move around several times during the temporal scope, and the various parent groups could change their types often. It uses the helper function `getTypes` (in Figure 4.24), which takes a group or feature entry and an interval, and returns the types the feature or group has during the interval.

If all the premises are true, then the FEATURES map is updated at `featureID` by shortening the interval key for the original type at t_n using `clampInterval`, and assigning the new type to the temporal scope $[t_n, t_{t_2})$.

4.8 Analysis Rule for Changing the Variation Type of a Group

The operation **changeGroupVariationType** changes the type of a group at a given time point, as defined in Section 3.2. The rule in Figure 4.25 is similar to the **changeFeatureVariationType** rule in Figure 4.23, and shows the semantics of changing the type of a group. In a similar way to the CHANGE-FEATURE-VARIATION-TYPE rule, it verifies that the types of all the child features during the affected interval are compatible with the new group type. If they are compatible, the group entry is updated with the new type during the temporal scope.

(CHANGE-GROUP-VARIATION-TYPE)

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

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

$$\text{GROUPS}\,[\texttt{groupID}] = \left(G_e,\, G_t,\, G_p,\, G_c\right)$$

---

changeGroupVariationType $(\texttt{groupID}, \texttt{type})$ at $t_n \triangleright$
$(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$
$\longrightarrow$
$(\text{NAMES}, \text{FEATURES},$
$\text{GROUPS}\,[\texttt{groupID}] \leftarrow \left(G_e,\, \texttt{clampInterval}(G_t, t_n)\,[[t_n, t_{t_2})] \leftarrow \texttt{type},\, G_p,\, G_c\right))$

Figure 4.25: The **CHANGE-GROUP-VARIATION-TYPE** rule

(CHANGE-FEATURE-NAME)

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

---

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

Figure 4.26: The **CHANGE-FEATURE-NAME** rule

4.9 Analysis Rule for Changing the Name of a Feature

As defined in Section 3.2, the **changeFeatureName** operation changes the name of a feature at a given time point. The semantics of changing the name of a feature are shown in the CHANGE-FEATURE-NAME rule in Figure 4.26. The old name and the next planned name change are identified on the first line ($F_n\,[t_n] = \{\texttt{oldName}\}$ and $F_n\,[t_n]_{\leq} = \{[t_{n_1}, t_{n_2})\}$ respectively). Since the name must not be in use during the temporal scope, we verify that looking up the new name in the NAMES map returns an empty set. The NAMES map is updated by shortening the interval for the old name to end at t_n, and assigning the feature ID to the new name during the temporal scope. Furthermore, the FEATURES map is updated at the feature ID, shortening the interval for the old name and assigning the new name to the temporal scope.

Chapter 5

Soundness

In this chapter we prove soundness for the analysis rules (Section 4 on page 37) by formalizing soundness for interval-based feature models and examining the behaviour of each rule individually.

Our goal is to show that applying a rule will result in a sound plan, and that the rules operate within the operation's scope and updates the model correctly. We first define formally what constitutes a well-formed interval-based feature model. Next, we prove that each rule results in a sound IBFM, given that the original IBFM is well-formed. Moreover, we show that the rules do not violate the scopes defined in Section 3.3, and that they update the model according to the semantics of each rule. We present only two of the proofs in this chapter to show the structure of the proofs. The rest of the proofs can be found in Appendix A on page 85.

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 well-formed. The soundness analysis in this chapter assumes that the original plan is sound; i.e., containing no paradoxes.

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 well-formed.

The well-formedness requirements listed in Section 2.1.1 on page 11 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 constant $RootID$ refers to the root of the interval-based feature model, and that $\text{FEATURES}\,[RootID] = (R_e, R_n, R_t, R_p, R_c)$. 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 child groups. 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 child group 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 $\texttt{featureID} \neq RootID$, 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 $F_t[t_n] = \{\text{OPTIONAL}\}$.

Since our representation does not enforce structural requirements, 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 the Rules

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) **Modularity** — proving that the rule operates strictly within the previously defined temporal and spatial scopes (see Section 3.3),

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

(iii) **Correctness of model modification** — 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 38 for the **ADD-FEATURE** rule. Let

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

be the initial state, and

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

the state after applying the **ADD-FEATURE** rule. 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***.

Modularity Recall from Section 3.3 on page 30 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.

Based on the above proof, we conclude with the following lemma:

Lemma 5.1. The **ADD-FEATURE** rule operates strictly within the temporal and spatial scopes of the **addFeature** operation.

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

Since the rule checks that the feature does not already exist during the temporal scope, it is impossible that `featureID` $= RootID$. Thus the rule does not affect the root feature, and ***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 child groups 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. Because of this, and since no feature uses the name during the temporal scope in the original plan, 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 child groups 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 child groups, 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 page 62), 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 any point t_k during the temporal scope 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 3.1 on page 20), and lookup in interval maps (Definition 3.3 on page 22), for all points t_k with $t_n \leq t_k < t_m$,

$$\text{NAMES}'\,[\texttt{name}]\,[t_k] = \{\texttt{featureID}\}$$

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 on page 19) and `setFeatureAttributes` (Figure 4.3 on page 39), and given that $\text{FEATURES}'\,[\texttt{featureID}] = \left(F'_e, F'_n, F'_t, F'_p, F'_c\right)$, then for all time points t_k

with $t_n \leq t_k < t_m$,

$$t_k \in_{\leq} F'_e \quad \text{the feature exists} \tag{1}$$
$$F'_n[t_k] = \{\texttt{name}\} \quad \text{the feature has the expected name} \tag{2}$$
$$F'_t[t_k] = \{\texttt{type}\} \quad \text{the feature has the expected type} \tag{3}$$
$$F'_p[t_k] = \{\texttt{parentGroupID}\} \quad \text{the feature has the expected parent group} \tag{4}$$
$$F'_c[t_k] = \emptyset \quad \text{the feature has no child groups} \tag{5}$$

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 $[t_n, t_m) \not\in_{\leqq} F_e$, 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 map is set to $\emptyset$ by `setFeatureAttributes`, 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 $\text{GROUPS}'[\texttt{parentGroupID}] = \left(G'_e, G'_t, G'_p, G'_c\right)$, then for all t_k with $t_n \leq t_k < t_m$,

$$\texttt{featureID} \in \bigcup G'_c[t_k]$$

meaning that the feature is in the parent group's set of child features in the updated model during the entire temporal scope. This holds because $\text{GROUPS}[\texttt{parentGroupID}]$ is assigned

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

which modifies G_c by adding `featureID` to the set of child features at interval key $[t_n, t_m)$ [1].

The above proof shows the following lemma:

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

[1] See Figure 4.4 on page 39 for the definition of `addChildFeature`.

5.2.2 Soundness of the Move Feature Rule

See Figure 4.19 on page 50 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`.

Modularity Recall that the temporal scope of the move-feature rule is $[t_n, t_k)$ (Section 3.3 on page 30), 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 50, since this is also referenced in the rule by $\neg\texttt{createsCycle}$. 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 ancestors and target group's ancestors, as well as the feature and the group themselves, and so the rule operates within the defined spatial scope.

Based on the above proof, we conclude with the following lemma:

Lemma 5.4. The MOVE-FEATURE rule operates strictly within the spatial and temporal scopes 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 child groups 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

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

As discussed in earlier sections (e.g. Section A.2 on page 87), `clampInterval` replaces a mapping $[\,[t_i, t_j) \mapsto v\,]$ by $[\,[t_i, t_n) \mapsto v\,]$, with $t_n \leq t_j$. The feature has no parent group after the application of $\texttt{clampInterval}(F_p,\, t_n)$. 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` (Figure 4.4) 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

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

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.

Since the rule adds a child feature to the target group during the temporal scope, the group must exist during the temporal scope for ***IBFM8*** to hold. The premise $[t_n, t_{p_2}) \in_{\leq} G_e$ along with the assumption that ***IBFM8*** holds in the origial plan ensure this. Moreover, the rule does not alter the feature's existence set, so ***IBFM8*** is preserved.

The intention of the cycle detection algorithm in Section 4.5.1 on page 50 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***.

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

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 child features during the temporal scope

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

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

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

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 all 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 preceding 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 child features during the temporal scope, and is in the new parent group's set of child features.

The above proof shows the following lemma:

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

In this section, we summarise the lemmas proven in Section 5.2 on page 61 and Appendix A on page 85 into three theorems that show that the rule system is sound — given a sound plan and an operation, if the operation is applied, then it causes no paradoxes, and the resulting model is well-formed. We base this on the definition of well-formedness for interval-based feature models listed in well-formedness requirements ***IBFM1–9***. Furthermore, the rules are modular, meaning that they operate within limited scopes.

In Section 5.2 and Appendix A, we have shown that each of the analysis rules operates within the temporal and spatial scopes of an operation. Based on the lemmas[2], we prove the following theorem.

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

We have shown that each of the analysis rules preserves well-formedness of the interval-based feature model (Section 5.2 and Appendix A). The following theorem follows directly from lemmas[3].

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

[2]See lemmas 5.1, 5.4, A.1, A.4, A.7, A.10, A.13, A.16, A.19
[3]See lemmas 5.2, 5.5, A.2, A.5, A.8, A.11, A.14, A.17, A.20

In Section 5.2 on page 61 and Appendix A, we have shown that each of the analysis rules behaves according to the semantics of the operation in question.[4] Using the lemmas proven there, we conclude that this holds for the rule system.

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

Together, the three theorems show that the rule system is sound. If applied to a sound interval-based feature model and an operation, they will update the model correctly if the operation does not cause a paradox (Theorem 5.9), and the resulting model will be well-formed (Theorem 5.8). The rules will only check the parts of the plan that may be structurally violated due to the change (Theorem 5.7).

[4] See lemmas 5.3, 5.6, A.3, A.6, A.9, A.12, A.15, A.18, A.21

Chapter 6

Implementation

In this chapter, we present our implementation of the data structures and analysis rules. 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] `https://github.com/idamotz/Master/tree/master/soundness-checker`
[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 (see Chapter 3 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 3.5 on page 23). 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 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`

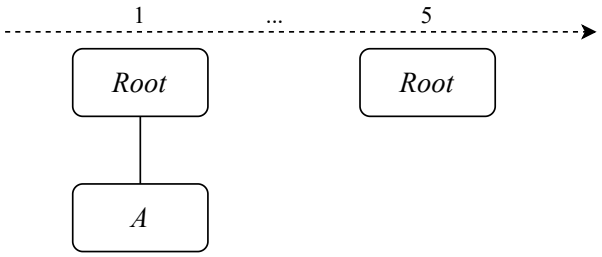


Figure 6.1: Simple plan

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.

6.1.2 Example — Encoding the Interval-Based Feature Model

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).

$$
\begin{aligned}
&(\{\,[\,\text{Root} \mapsto [\,[1,\infty) \mapsto \text{feature:root}\,]\,]\}\\
&\\
&,\{\,[\text{feature:root} \mapsto\\
&\qquad\qquad (\{[1,\infty)\},\\
&\qquad\qquad \{[\,[1,\infty) \mapsto \text{Root}\,]\},\\
&\qquad\qquad \{[\,[1,\infty) \mapsto \textsc{mandatory}\,]\},\\
&\qquad\qquad \emptyset,\\
&\qquad\qquad \{[\,[1,5) \mapsto \text{group:A}\,]\})]\\
&\;\}\\
&\\
&,\{\,[\text{group:A} \mapsto\\
&\qquad\qquad (\{[1,5)\},\\
&\qquad\qquad \{[\,[1,5) \mapsto \textsc{and}\,]\},\\
&\qquad\qquad \{[\,[1,5) \mapsto \text{feature:root}\,]\},\\
&\qquad\qquad \emptyset)]\\
&\;\})
\end{aligned}
$$

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

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

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 update operation is simply called an `UpdateOperation`. To express **addFeature**(feature:B, B, MANDATORY, group:A) from 3 to ∞, we write

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

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

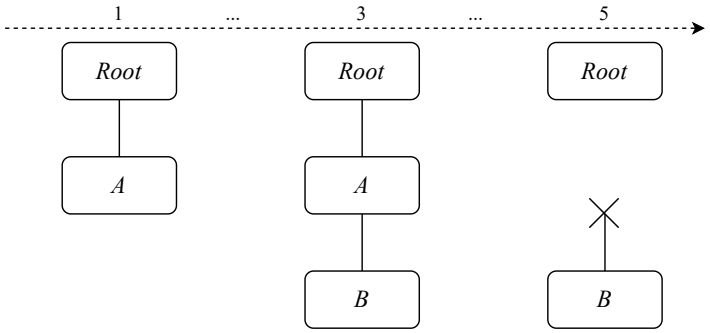


Figure 6.2: Illustration of the paradox

6.1.3 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 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 `UpdateOperation` 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 identified quite precisely.

To apply the operations, the `Apply` module exports the function `apply`, which takes a `UpdateOperation` 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 does not result in an error, we will 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 example, 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.

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 necessary for modifying a feature model evolution plan?* We have chosen a set of update operations (i.e. **addFeature(...)** from t_n to t_m) to update a feature model evolution plan (see Section 3.2 on page 28). 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 and formalize a feature model evolution plan in such a way that the scope of each operation can be captured?* The representation we have devised — the interval-based feature model, defined in Section 3.1 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 soundly analyse change?* We have given rules for analysis of change in Section 4 on page 37. The rules rely on the assumption that the initial plan is sound, and as proven in Chapter 5 on page 59, they do guarantee that the resulting plan is sound if a rule can be applied. Furthermore, the prototype[1] 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

[1] `https://github.com/idamotz/Master/tree/master/soundness-checker`

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 of the Add Group Rule

See Figure 4.7 on page 45 for the **ADD-GROUP** rule. Let

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

be the initial state, and

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

be the state after the **ADD-GROUP** rule is applied. Recall that this operation adds the group with ID `groupID` to the interval-based feature model $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ from t_n to t_m.

Modularity Recall from Section 3.3 on page 30 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` (see Figure 4.8 on page 45) and `setGroupAttributes` (Figure 4.9 on page 45) 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 temporal and spatial scopes 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.

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

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 GROUPS [groupID].

Looking up the added group's ID in the modified model during the temporal scope should return the information we put in the operation. Given $\text{GROUPS}'\,[\texttt{groupID}] = \left(G'_e,\, G'_t,\, G'_p,\, G'_c\right)$, then for all time points t_k with $t_n \leq t_k < t_m$, the following statements hold:

$$t_k \in_{\leq} G'_e \qquad \text{the group exists} \tag{1}$$

$$G'_t\,[t_k] = \{\texttt{type}\} \qquad \text{the group has the expected type} \tag{2}$$

$$G'_p\,[t_k] = \{\texttt{parentFeatureID}\} \qquad \text{the group has the expected parent feature} \tag{3}$$

$$G'_c\,[t_k] = \emptyset \qquad \text{the group has no children} \tag{4}$$

Statement (1) holds due to the line $G_e \cup \{[t_n, t_m)\}$ in `setGroupAttributes` (Figure 4.9 on page 45). 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 $\text{FEATURES}'\,[\texttt{parentFeatureID}] = \left(F'_e,\, F'_n,\, F'_t,\, F'_p,\, F'_c\right)$, then

$$\texttt{groupID} \in \bigcup F'_p\,[t_k]$$

for all t_k with $t_n \leq t_k < t_m$. In the **ADD-GROUP** rule,

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

is assigned to FEATURES [parentFeatureID]. The function `addChildGroup` (Figure 4.8 on page 45) 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 during the temporal scope.

The above proof shows the following lemma:

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 4.10 on page 47 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, and

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

be the state after the **REMOVE-FEATURE** rule has been applied. Recall that this operation removes the feature with ID `featureID` from the interval-based feature model (NAMES, FEATURES, GROUPS) at t_n. Furthermore, let 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.

Modularity As defined in Section 3.3 on page 30, 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 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 4.11 on page 48), 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 4.14 on page 48), 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 4.16 on page 48) modifies the parent group's set of child features 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 child features, and adds the feature to the set of child features 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 child groups. The premise

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

ensures that the operation is not applied unless the feature's set of child groups 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.

Based on the above proof, we conclude with the following lemma:

Lemma A.4. The **REMOVE-FEATURE** rule operates strictly within the temporal and spatial scopes of the **removeFeature** operation.

Preserving well-formedness Let

$$\text{FEATURES}'\,[\texttt{featureID}] = \left(F'_e,\, F'_n,\, F'_t,\, F'_p,\, F'_c\right)$$

The **REMOVE-FEATURE** rule contains the premise

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

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, $t_n \not\leq F'_e$ due to the semantics of `clampFeature` (Figure 4.14 on page 48), so ***IBFM3*** holds trivially for the updated model. The only change made to a group is by the function `removeFeatureAt` (Figure 4.16 on page 48), which removes the

feature from the parent group's map of child groups 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 $F_c\,[[t_n, t_{e_2})] = \emptyset$ ensures that the feature to be removed does not have any child groups 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 $[t_n, t_j) \not\trianglelefteq F'_e$, 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 $\texttt{featureID} \notin \text{NAMES}\left[\texttt{name}\right]\left[[t_n, t_j)\right]$. The former statement holds due to the semantics of `clampFeature` and `clampInterval`; the feature's attributes are all clamped to end at the time of removal, and the premises on the form $F_x\,[[t_n, t_{e_2})] = \{v\}$ 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 $[\,[t_i, t_j) \mapsto \texttt{featureID}\,]$ is replaced by $[\,[t_i, t_n) \mapsto \texttt{featureID}\,]$ in the function $\texttt{clampInterval}(\text{NAMES}\left[\texttt{name}\right], t_n)$. 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.

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

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 $\text{FEATURES}'\,[\texttt{featureID}] = (e,\ n,\ t,\ p,\ c)$, and $F_e\,[t_n]_{\leq} = [t_i, t_j)$, then

$$[t_n, t_j) \notin_{\lessapprox} F_e \qquad \text{the feature does not exist} \tag{1}$$
$$F_n\left[[t_n, t_j)\right] = \emptyset \qquad \text{the feature has no name} \tag{2}$$
$$F_t\left[[t_n, t_j)\right] = \emptyset \qquad \text{the feature has no type} \tag{3}$$
$$F_p\left[[t_n, t_j)\right] = \emptyset \qquad \text{the feature has no parent group} \tag{4}$$
$$F_c\left[[t_n, t_j)\right] = \emptyset \qquad \text{the feature has no child groups} \tag{5}$$

Since we established $[t_n, t_j) \notin_{\lessapprox} F_e$ in the above paragraph, these statements follow directly from Lemma A.5 on the preceding page 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 child feature.

The above proof shows the following lemma:

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 4.18 on page 49 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 $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ at t_n. Furthermore, let $\text{GROUPS}\,[\texttt{groupID}] = (G_e,\ G_t,\ G_p,\ G_c)$, 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.

Modularity As defined in Section 3.3 on page 30, 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 4.15 on page 48), 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 $G_t\,\left[[t_n, t_j)\right] = \{\texttt{type}\}$ and $G_p\,\left[[t_n, t_j)\right] = \{\texttt{parentFeatureID}\}$, 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 4.17 on page 48) modifies the parent feature's set of child groups 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 child groups 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 child features. The premise

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

ensures that the operation is not applied unless the group's set of child features 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.

Based on the above proof, we conclude with the following lemma:

Lemma A.7. The **REMOVE-GROUP** rule operates strictly within the temporal and spatial scopes of the **removeGroup** operation.

Preserving well-formedness Let

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

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 child group 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 child groups — 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 ($[t_n, t_j) \not\in_{\leqq} G'_e$), 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 child features 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, $[t_n, t_j) \not\in_{\leqq} G'_e$, 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 $G_t\,[[t_n, t_{e_2})] = \{\texttt{type}\}$ in **ADD-GROUP**, we have that

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

Clearly, $G_t''\,[[t_n, t_j)] = \emptyset$. Furthermore, since $[t_{t_1}, t_n)$ does not overlap $[t_n, t_j)$, $G_t'\,[[t_n, t_j)] = \emptyset$. An analogous argument can be made for $G_p'\,[[t_n, t_j)] = \emptyset$. From the definition of clampGroup, $G_c = G_c'$, so by the premise $G_c\,[[t_n, t_j)] = \emptyset$ in the rule, $G_c'\,[[t_n, t_j)] = \emptyset$. 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.

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

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 child features 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.

The above proof shows the following lemma:

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 4.22 on page 53 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`.

Modularity Recall that the temporal scope of the **MOVE-GROUP** rule is $[t_n, t_k)$ (Section 3.3 on page 30), 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 50, 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 ancestors and the target feature's ancestors, as well as the group and feature themselves, and so the rule operates within the defined spatial scope.

Based on the above proof, we conclude with the following lemma:

Lemma A.10. The **MOVE-GROUP** rule operates strictly within the temporal and spatial scopes 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 child group 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 child features map are not modified, ***IBFM4–5*** and ***IBFM7*** are true for the modified model.

Since the rule adds a child group to the target feature during the temporal scope, the feature must exist during the temporal scope for ***IBFM8*** to hold. The premise $[t_n, t_{p_2}) \in_{\leq} F_e$ along with the assumption that ***IBFM8*** holds in the origial plan ensure this. Moreover, 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 50 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***.

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

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 child groups during the temporal scope

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

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

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

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 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 child groups during the temporal scope, and is in the new parent feature's set of child groups.

The above proof shows the following lemma:

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 4.23 on page 54 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`.

Modularity Recall that the temporal scope of the **CHANGE-FEATURE-VARIATION-TYPE** rule is $[t_n, t_k)$ (Section 3.3 on page 30), 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

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

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.

Based on the above proof, we conclude with the following lemma:

Lemma A.13. The **CHANGE-FEATURE-VARIATION-TYPE** rule operates strictly within the temporal and spatial scopes of the **changeFeatureVariationType** operation.

Preserving well-formedness Due to the premise $\texttt{featureID} \neq \mathit{RootID}$, 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

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

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.

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

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 4.25 on page 56 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`.

Modularity Recall that the temporal scope of the **CHANGE-GROUP-VARIATION-TYPE** rule is $[t_n, t_k)$ (Section 3.3 on page 30), 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 child features during the temporal scope, which may both move and change their types. These features and

their types are looked up in the premise

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

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 temporal and spatial scopes 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 child feature mappings overlapping the temporal scope ($[\,[t_{c_1}, t_{c_2}) \mapsto \{f_1, f_2, \ldots\}\,]$), finds the types each child feature has during the scope and *while* it is the child feature 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 4.26 on page 56 for the **CHANGE-FEATURE-VARIATION-TYPE** rule. Let

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

be the initial state, and

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

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

Modularity Recall that the temporal scope of the **CHANGE-FEATURE-NAME** rule is $[t_n, t_k)$ (Section 3.3 on page 30), 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.

Based on the above proof, we conclude with the following lemma:

Lemma A.19. The **CHANGE-FEATURE-NAME** rule operates strictly within the temporal and spatial scopes 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

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

Here, the interval containing t_n in NAMES [oldName] 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.

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

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 $\text{FEATURES}'\,[\texttt{featureID}] = \left(F'_e, F'_n, F'_t, F'_p, F'_c\right)$ 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 $t_n \leq t_i < t_{n_2}$,

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

Additionally, we should have $\text{NAMES}'[\texttt{name}] = \texttt{featureID}$. 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 **changeFeatureName** operation.