

Master Thesis

Methods for modular soundness checking of feature model evolution plans

Ida Sandberg Motzfeldt



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

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Master Thesis

*Methods for modular soundness checking
of feature model evolution plans*

Ida Sandberg Motzfeldt

© 2021 Ida Sandberg Motzfeldt

Master Thesis

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

Printed: Reprosentralen, University of Oslo

Abstract

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

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

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

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

TODO: Citing: it's okay not to cite in the abstract, but remember to back up everything in the introduction

Contents

I	Introduction and Background	1
1	Introduction	2
1.1	Research questions	3
1.2	Contributions	3
2	Background	4
2.1	Software product lines	4
2.1.1	Feature models	4
2.1.2	Feature model evolution plans	5
2.2	Static analysis	7
2.2.1	Analysis context (rules)	9
2.2.2	Soundness	9
II	The project	10
3	Definitions and semantics	12
3.1	Definitions	12
3.1.1	Interval-based feature model	12
3.1.2	Maps	13
3.1.3	Intervals	13
3.1.4	Interval maps	14
3.1.5	Interval sets	15
3.1.6	Mapping names	15
3.1.7	Mapping features	15
3.1.8	Mapping groups	16
3.1.9	Example evolution plan	16
3.1.10	Operations	18
3.2	Define the scope	19
3.3	SOS rules	22
3.3.1	Add feature rule	22
3.3.2	Add group rule	24
3.3.3	Remove feature rule	24
3.3.4	Remove group rule	25

3.3.5	Algorithm for detecting cycles resulting from move operations	27
3.3.6	Move feature rule	28
3.3.7	Move group rule	29
3.3.8	Change feature variation type rule	30
3.3.9	Change group variation type rule	30
3.3.10	Change feature name	32
4	Soundness	33
4.1	Soundness for interval-based feature models	33
4.2	Soundness of each rule	36
4.2.1	Soundness of the Add feature rule	36
4.2.2	Soundness for the Add group rule	39
4.2.3	Soundness of the Remove feature rule	42
4.2.4	Soundness of the Remove group rule	45
4.2.5	Soundness of the Move feature rule	48
4.2.6	Soundness of the Move group rule	51
4.2.7	Soundness of the Change feature variation type rule	54
4.2.8	Soundness of the Change group variation type rule	56
4.2.9	Soundness of the Change feature name rule	58
4.3	Soundness of the rule system	60
5	Implementation	61
III	Conclusion	62

List of Figures

2.1	Example feature model for a coffee machine	4
2.2	Coffee machine with added touch interface	6
2.3	Formalized feature model evolution plan	7
3.1	Interval map example	14
3.2	Small interval-based feature model	17
3.3	The ADD-FEATURE SOS rule	23
3.4	compatibleTypes	23
3.5	setFeatureAttributes	23
3.6	addChildFeature	24
3.7	The ADD-GROUP SOS rule	24
3.8	setGroupAttributes	24
3.9	addChildGroup	25
3.10	The REMOVE-FEATURE SOS rule	25
3.11	clampInterval	26
3.12	clampIntervalValue	26
3.13	clampIntervalSet	26
3.14	clampFeature	26
3.15	clampGroup	26
3.16	removeFeatureAt	26
3.17	removeGroupAt	26
3.18	The REMOVE-GROUP SOS rule	27
3.19	ancestors	28
3.20	The MOVE-FEATURE SOS rule	29
3.21	The MOVE-GROUP SOS rule	30
3.22	The CHANGE-FEATURE-VARIATION-TYPE SOS rule	31
3.23	getTypes	31
3.24	The CHANGE-GROUP-VARIATION-TYPE SOS rule	31
3.25	The CHANGE-FEATURE-NAME SOS rule	32

List of Tables

Preface

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

SPLs grow large as they are more profitable the more variants they originate [1], and evolve over time as requirements change. For complex projects, planning is necessary, and these plans, too, change with requirements. **TODO: Si noe mer om hva planlegging innebærer eller betyr på en intuitiv måte før jeg går over til den tekniske forklaringen** The research project this thesis is built upon defines *feature model evolution plans* (FMEP) in terms of feature models [3]. **TODO: Belys en forskjell på en plan og en planendring** A FMEP consists of the initial feature model and a sequence of edit operations. In the same project, we have devised a method to detect paradoxes in a feature model evolution plan. **TODO: Beskriv paradoxes** However, when a plan is changed, this method requires that the entire plan be verified, and for large plans, this may be inefficient. This is due to the fact that the method relies on feature model change rather than FMEP change.

1.1 Research questions

RQ1 Which operations are useful for modifying a feature model evolution plan?

RQ2 How can we capture all the states of a feature model evolution plan in such a way that the scope of each operation can be isolated?

RQ3 How can we ensure soundness of change?

TODO: Describe the goals in more detail **TODO:** It should be clear to the reader that I have soundness proofs

1.2 Contributions

In this thesis, we model and create an analysis method for validating *plan* change, with the goal that only the parts of the plan which may be affected by that change are checked for paradoxes. This method may then be used as basis for an SPL planning tool. We define a representation suited for local lookup and assignment **TODO: of and for what?**, and create analysis rules for safely updating the evolution plan. We implement the method in a prototype, and give a proof that the method is sound.

Chapter 2

Background

TODO: Meeting notes

2.1 Software product lines

TODO: Write yet *another* introduction to software product lines..

2.1.1 Feature models

To model the relations between the features (which combinations are allowed, which are common to all variants, etc.), it is common to use a *feature model*. A feature model is a tree-like structure where the nodes are features and groups of features. See Figure 2.1 for an example of a feature

¹Created using DarwinSPL: <https://gitlab.com/DarwinSPL/DarwinSPL>

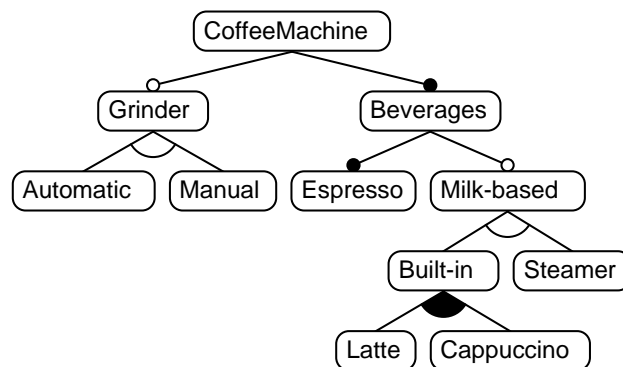


Figure 2.1: Example feature model for a coffee machine ¹

model. A feature cannot be selected in a variant unless its parent feature is also selected. [2]

A group gives logical structure to the features, restricting the allowed combinations of the features. For instance, in an *alternative* group, exactly one of the features must be selected in every variant. In the example, the group under Grinder has this type. In a variant, a grinder cannot be both automatic and manual. Moreover, the features have types (OPTIONAL and MANDATORY). A MANDATORY feature must be selected in all variants, whereas an OPTIONAL feature may be left out. The black dot above Beverages means that this feature is mandatory, so all coffee machines provide beverages. Furthermore, since its subfeature Espresso is also mandatory, all coffee machines have espresso. However, only some coffee machines have milk-based drinks, as shown by the white dot above the Milk-Based feature. If selected, then either built-in drinks such as latte or cappuccino must be included in the variant, or the machine must have a steamer so the user can make milk-based drinks themselves. The group under Built-in is filled-in with black, which means that it is an OR group. In a variant where Built-in is chosen, one or both of Latte and Cappuccino must also be chosen, but not zero. The groups which are neither ALTERNATIVE nor OR have the type AND, which means that zero, one, or more of its subfeatures may be selected in a variant. There are several restrictions to the structure of a feature model. For instance, an ALTERNATIVE or OR group cannot contain a MANDATORY feature. The root feature (here CoffeeMachine) must have type MANDATORY, since naturally it must be selected in all variants.

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

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 exists tools for evolution planning, such as DarwinSPL². An

²<https://gitlab.com/DarwinSPL/DarwinSPL>

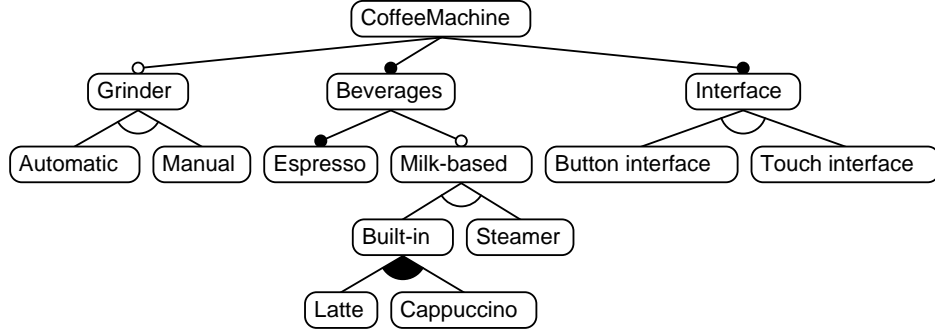


Figure 2.2: Coffee machine with added touch interface ³

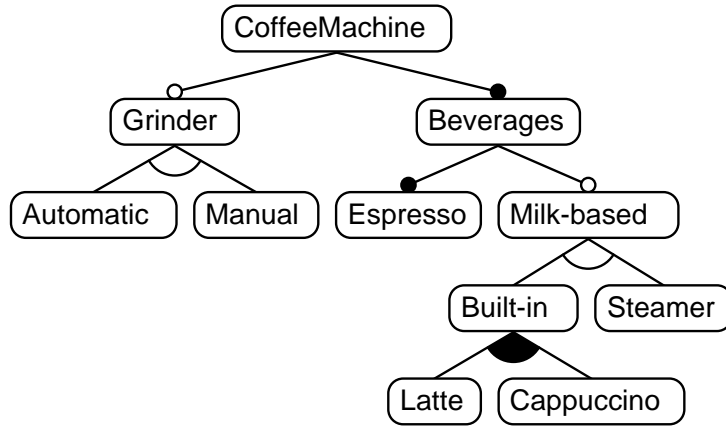
intuitive way to think of an evolution plan is a sequence of feature models associated with the time points when they are planned to be realized. For instance, imagine that the first coffee machines in the software product line are represented by the feature model in Figure 2.1 on page 4. They are controlled by buttons, but as touch interfaces become more common, the manager decides to add coffee machines with a touch pad. Furthermore, voice commands are becoming trendy, and she feels that the feature model should have options for controlling it by voice, too. 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 (points in time **TODO: hm**) 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 identify the features and groups to be added, removed, or modified. 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 4. Applying the operations under t_1 results in the feature model shown in Figure 2.2. In the examples, IDs are not used, and the operations are simplified for readability. In the formal definitions, such details are included in the operations.

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

³Created using DarwinSPL: <https://gitlab.com/DarwinSPL/DarwinSPL>

t_0



- t_1
- Add new MANDATORY feature “Interface” to “CoffeeMachine” AND group
 - Add new ALTERNATIVE group to “Interface”
 - Add new feature “Button interface” to “Interface” ALTERNATIVE group
 - Add new feature “Touch interface” to “Interface” ALTERNATIVE group

Figure 2.3: Formalized feature model evolution plan

model evolution plan.

2.2 Static analysis

Static analysis attempts to predict the behaviour of a program without executing it. Due to the halting problem, it is impossible for an algorithm to accomplish this completely, but there exist several methods to safely approximate information. For instance, live variable analysis discovers which variables *may* still be “alive” (meaning used in the future) at a certain point of the program. Live variable analysis is done for optimization of memory allocation for variables. If a variable is known not to be live, it is safe to overwrite it with another variable. If a variable *may* be live, it should not be overwritten. This makes live variable analysis a *may analysis*. It is also a backward analysis, since information about whether a variable is used in the future is carried backwards. There also exist *must analyses* and *forward analyses*. A must analysis looks for the greatest solution of things that *must* be true. In a forward analysis, the information flows forward; meaning that what has happened earlier in the program influences the analysis at later points in the program.

Unlike programs, it is possible to get full overview of a feature model evolution plan. It is always possible to find the correct answer given the question “Does feature A exist at time 5?”. An operational representation

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

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

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

, 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

$$\frac{\Gamma[b] = \top}{\langle \Gamma, \text{if } b \text{ then } S \text{ else } S' \rangle \longrightarrow \langle \Gamma, S \rangle} \text{IF}_1 \quad \frac{\Gamma[b] = \perp}{\langle \Gamma, \text{if } b \text{ then } S \text{ else } S' \rangle \longrightarrow \langle \Gamma, S' \rangle} \text{IF}_2$$

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 \perp (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 there is no rule that defines the program behaviour if value of b is neither \top nor \perp . This means that anything other than \top or \perp 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. 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 Analysis context (rules)

2.2.2 Soundness

TODO: Reread essay and identify useful parts

Part II

The project

TODO: Rename this part

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

TODO: Need to structure the project part more finely

Chapter 3

Definitions and semantics

TODO: Explain why it's non-trivial, why it may be difficult to do it manually, why we need to restrict the scope etc. Why simple lookup is not enough. TODO: First, explain what the challenge is with doing modular checking. Why is it difficult, why I pay attention to it, how does it affect the rules I have created? TODO: Use examples!!!

3.1 Definitions

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

3.1.1 Interval-based feature model

Definition 3.1 (Interval-based feature model). An *interval-based feature model* is defined as a triple $(\text{NAMES}, \text{FEATURES}, \text{GROUPS})$ where NAMES is a map from *names* to *feature IDs*, FEATURES is a map from *feature IDs* to *feature entries*, and GROUPS is a map from *group IDs* to *group entries*.

The reason for this choice is efficiency and modularity. As mentioned in **TODO: motivation or background or something**, the goal of this thesis is to minimize the scope of the plan to check for paradoxes, as a change rarely affects more than a small part of the plan. It would then be a mistake to represent a plan as a sequence of trees associated with time points, or an initial model followed by a sequence of operations as described in "Consistency-preserving evolution planning on feature models" [3]. To add a new feature to the plan, both representations would require us to look through the entire plan to check that the feature ID is fresh.

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

3.1.2 Maps

Definition 3.2 (Map). A *map* is a set of entries on the form $[k \mapsto v]$, where each key k uniquely defines a value v .

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

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

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

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

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

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

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

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

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

3.1.3 Intervals

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

Definition 3.3 (Interval). An interval is a set of time points between a lower and an upper bound. We denote the interval using the familiar mathematical notation $[t_{\text{start}}, t_{\text{end}})$, where t_{start} is the lower bound, and t_{end} is the upper bound. These intervals are left-closed and right-open, meaning that t_{start} is contained in the interval, and all time points until but not including t_{end} .

We say that an interval $[t_{\text{start}}, t_{\text{end}})$ *contains* the time point t_k if $t_{\text{start}} \leq t_k < t_{\text{end}}$. Two intervals $[t_n, t_m)$ and $[t_i, t_j)$ *overlap* if there exists a time point t_k with $t_n \leq t_k < t_m$ and $t_i \leq t_k < t_j$, i.e. a time point contained by both intervals.

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

3.1.4 Interval maps

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

To look up values, one can either give an interval or a time point as key. Both will return sets of values. For instance, if an interval map I contains the mapping $[t_1, t_5) \mapsto v$, all of the queries in Figure 3.1 will return $\{v\}$ (assuming that $t_1 < t_2 < \dots < t_5$):

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

Figure 3.1: Interval map example

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

We furthermore define function $IM[[t_n, t_m)]_{\geq}$ which returns all the interval keys in the map IM overlapping the interval $[t_n, t_m)$.

3.1.5 Interval sets

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

Given an interval set IS , $[t_n, t_m) \in IS$ if $[t_n, t_m)$ is a member of the set, which is the expected semantics of \in . We define a similar predicate \in_{\leq} such that $[t_n, t_m) \in_{\leq} IS$ iff there exists some interval $[t_i, t_j) \in IS$ with $t_i \leq t_n \leq t_m \leq t_j$, i.e. an interval in IS which contains $[t_n, t_m)$. We further define the predicate \in_{\subseteq} such that $[t_n, t_m) \in_{\subseteq} IS$ iff there exists some interval $[t_i, t_j) \in IS$ with $[t_n, t_m)$ overlapping $[t_i, t_j)$.

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

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

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

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

3.1.6 Mapping names

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

3.1.7 Mapping features

The FEATURES map has entries of the form $[\text{featureID} \mapsto \text{feature entry}]$. Since several pieces of information are crucial to the analysis of a feature, it is not enough to have a simple mapping as we have for names. A feature has a name, a type, a parent group, and zero or more child groups. Furthermore, a feature may be removed and re-added during the course

of the plan, so we also need information about when the feature exists. This information is collected into a 5-tuple $(F_e, F_n, F_t, F_p, F_c)$, where F_e is an interval set denoting when the feature exists, F_n is an interval map with name values, F_t is an interval map with the feature's variation types, F_p is an interval map with group ID values, and F_c is an interval map where the values are sets containing group IDs, the interval keys possibly overlapping.

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

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

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

3.1.8 Mapping groups

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

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

3.1.9 Example evolution plan

A small example of an interval-based feature model can be found in Figure 3.2 on the next page. It contains three features and one group, and describes an interval-based feature model for a washing machine. The washing machine always has a washer, and a dryer is added at t_5 . It is clear from this example that the representation is better suited for manipulating the structure than reading it.

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

Figure 3.2: Small interval-based feature model

3.1.10 Operations

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

- **addFeature(featureID, parentGroupID, name, featureType)** from t_n to t_m
Adds feature with id featureID, name name, and feature variation type featureType to the group with id parentGroupID in the interval $[t_n, t_m)$. featureID must be fresh, and the name cannot belong to any other feature in the model during the interval. The parent group must exist during the interval, and the types of the feature and the parent group must be compatible. If the feature has type MANDATORY, then the parent group must have type AND.
- **addGroup(groupID, parentFeatureID, groupType)** from t_n to t_m
Adds group with id groupID and type groupType to the feature with id parentFeatureID during the interval $[t_n, t_m)$. The group ID must be fresh, and the parent feature must exist during the interval.
- **removeFeature(featureID)** at time t_n
Removes the feature with ID featureID from the feature model at t_n (does not affect possible reintroductions). The FEATURES map in the original plan must contain a mapping $[featureID \mapsto (EXISTENCE, \dots)]$ such that $[t_i, t_j) \in EXISTENCE$ with $t_i \leq t_n \leq t_j$. The feature must not have any child groups during $[t_n, t_j)$. After removing the feature, all interval mappings should be updated from $[t_i, t_j)$ to $[t_i, t_n)$.
- **removeGroup(groupID)** at time t_n
Removes the group with ID groupID from the feature model at t_n (does not affect possible reintroductions). The GROUPS map in the original plan must contain a mapping $[groupID \mapsto (EXISTENCE, \dots)]$ such that $[t_i, t_j) \in EXISTENCE$ with $t_i \leq t_n \leq t_j$. The group must not have any child features during $[t_n, t_j)$. After removing the group, all interval mappings should be updated from $[t_i, t_j)$ to $[t_i, t_n)$.
- **moveFeature(featureID, targetGroupID)** at t_n
Moves the feature with id featureID to the group with ID targetGroupID. The move cannot be done if it introduces a cycle;

that is, if the target group is in the feature's subtree. The feature's subtree is moved along with it. Parent group and child feature mappings are updated accordingly.

- **moveGroup(groupID, targetFeatureID)**
Moves the group with id groupID to the feature with ID targetFeatureID. Very similar to **moveFeature**.
- **changeFeatureVariationType(featureID, newType)**
Changes the feature variation type of the feature with ID featureID to newType. If the new type is MANDATORY, the parent group type must be AND.
- **changeGroupVariationType(groupID, newType)**
Changes the group variation type of the group with ID groupID to newType. If the new type is OR or ALTERNATIVE, make sure that no child feature has type MANDATORY.
- **changeFeatureName(featureID, name)**
Changes the name of the feature with ID featureID to name. No other feature may have the same name.

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

TODO: move scope and operations above definitions

3.2 Define the scope

What is the scope?

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

Time

Which time points of the plan may be affected by the change?

Space

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

Operation scopes

We define the *minimal* scope for each operation.

- **addFeature**(featureID, parentGroupID, name, featureType) from t_n to t_m

We argue that the temporal scope is $[t_n, t_m)$, since this is the only interval in which the interval-based feature model is affected by the change. In other words, if we look at the interval-based feature model as a sequence of feature models, the only models that may break as a result of this modification, are the ones associated with time points between t_n and (but not including) t_m . The spatial scope must be only the feature itself, the parent group and the name. If the group type of the parent changes to a conflicting one, the operation is unsound. If the parent group is removed, we have an orphaned feature, which is also illegal. The name is unique, so we must also verify that no other feature is using the name during the temporal scope.

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

- **removeFeature**(featureID) at t_n

TODO: Explain why the temporal scope is the way it is If the original interval containing t_n in which the feature exists inside the feature model is $[t_m, t_k)$, then the temporal scope is $[t_n, t_k)$ - from the feature is removed until it would have been removed anyway
TODO: rephrase. Since the feature is removed at t_k in the original plan, and the original plan is sound as we assume, removing the feature earlier may only affect the plan in the interval between these two time points.

The spatial scope must be the feature itself, its parent group, and its subgroups. If the feature has or will have a subgroup during the interval, then it cannot be removed. Otherwise, there are no conflicts. When modifying the interval-based feature model, the feature must

be removed from the parent's set of subfeatures, which is why the parent group is included in the spatial scope.

- **removeGroup**(groupID) at t_n
Extremely similar to **removeFeature**.
- **moveFeature**(featureID, targetGroupID) at t_n
If t_m is the time at which the feature is next moved in the original plan, the temporal scope is $[t_n, t_m)$, since this operation only affects the plan within this interval.
The spatial scope is discussed in more detail in the **move feature algo**. This scope is the largest and hardest to define, because we have to detect cycles. The scope is defined by the feature and its ancestors, as well as target group and its ancestors, which may change during the intervals. It is not necessary to look at all ancestors, only the ones which feature and targetGroup do not have in common, as well as the feature and the group themselves. As usual, conflicting types and removal must be considered in addition to cycles.
- **moveGroup**(groupID, targetFeatureID) at t_n
See moveFeature. Very similar.
- **changeFeatureVariationType**(featureID, newType) at t_n
Temporal scope: $[t_n, t_m)$ if t_m is the next time point at which the feature's type changes or when feature is (next) removed.
Spatial scope: The only possibly conflicting thing in the feature model is the parent group's type. At no point must the feature have type 'mandatory' and the parent group have type 'alternative' or 'or'. Thus, the spatial scope is the parent group.
- **changeGroupVariationType**(groupID, newType) at t_n Temporal scope: Same as previous.
The spatial scope are the group's child features; the possible conflict is the same as with changeFeatureType.
- **changeFeatureName**(featureID, name) at t_n Temporal scope: Same as previous.
Spatial scope: The name, the feature, and its previous name. If it already exists within the feature model during the interval, or if the feature does not exist, then the change is invalid.

TODO: deal with batch operations/reverting a change: It is currently impossible to *extend* an interval; If a feature exists during $[t_3, t_5)$, it is impossible to change the plan such that it exists during $[t_3, t_6)$ instead. Don't really know how to fix that, except maybe adding an operation. In Figure 3.2 on page 17, if we try to change the name of feature 1 to

Dryer at t_2 , intending to change it back before Dryer is added, then these semantics will reject the first change, as two features will have the name Dryer during $[t_5, \infty)$. The paradox would be righted once we add that the name will change back to Washer at t_4 . There are workarounds for this, for instance changing the name of feature 2 to some temporary placeholder, making the changes to feature 1, and then changing feature 2 back. This, however, seems too cumbersome. Hopefully this use case is not common enough that most users will suffer for it, but it is definitely an example of the semantics being too strict.

3.3 SOS rules

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

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

TODO: Remember premises = above the line

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

SOS rules TODO: explain what SOS rules are used for

The rules are on the form

(RULE-LABEL)

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

where S is the state, and S' is the new state after the rule is applied. The rule can only be applied if all the premises hold. In this thesis, the state is always on the form **operation** \triangleright (NAMES, FEATURES, GROUPS), where **operation** denotes the change we intend to make to the interval-based feature model (NAMES, FEATURES, GROUPS). The new state is always on the form (NAMES', FEATURES', GROUPS'), where the maps have been updated according to the semantics of the operation. The premises ensure that an operation can only be applied if some conditions hold; for instance the **ADD-FEATURE** rule 3.3 contains premises verifying that the feature does not already exist when we wish to add it.

3.3.1 Add feature rule

Figure 3.3 describes the semantics of the **addFeature** operation. To add a feature during the interval $[t_n, t_m)$, its ID cannot exist during the

(ADD-FEATURE)

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

Figure 3.3: The ADD-FEATURE SOS rule

```

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

```

Figure 3.4: compatibleTypes

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

To make the rule tidier, we use three helper functions: `compatibleTypes` (Figure 3.4), `setFeatureAttributes` (Figure 3.5), and `addChildFeature` (Figure 3.6).

```

setFeatureAttributes((F_e, F_n, F_t, F_p, F_c), [t_start, t_end), name, type
                    , parentGroupID)
= ( F_e \cup [t_start, t_end)
  , F_n [[t_start, t_end)] \leftarrow name
  , F_t [[t_start, t_end)] \leftarrow type
  , F_p [[t_start, t_end)] \leftarrow parentGroupID
  , F_c )

```

Figure 3.5: setFeatureAttributes

$$\begin{aligned} & \text{addChildFeature}((G_e, G_t, G_p, G_c), [t_{start}, t_{end}], \text{fid}) \\ &= \left(G_e, G_t, G_p, G_c \left[[t_{start}, t_{end}] \right] \stackrel{\cup}{\leftarrow} \text{fid} \right) \end{aligned}$$

Figure 3.6: addChildFeature

(ADD-GROUP)

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

Figure 3.7: The ADD-GROUP SOS rule

3.3.2 Add group rule

The rule in figure 3.7 describes the conditions which must be in place to add a (pre-existing or fresh) group to the FMEP during an interval $([t_n, t_m))$. The group must not already exist in the plan during the interval $([t_n, t_m) \not\in_{\leq} G_e)$, and the parent feature must exist for the duration of the interval $([t_n, t_m) \in_{\leq} F_e)$. The group ID is added to the parent feature's map of child groups with the interval as key, and the attributes specified are added to the group entry in the GROUPS map.

3.3.3 Remove feature rule

Figure 3.10 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

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

Figure 3.8: setGroupAttributes

$$\begin{aligned} & \text{addChildGroup}((F_e, F_n, F_t, F_p, F_c), [t_{\text{start}}, t_{\text{end}}], \text{groupID}) \\ &= \left(F_e, F_n, F_t, F_p, F_c \left[[t_{\text{start}}, t_{\text{end}}] \right] \stackrel{\cup}{\leftarrow} \text{groupID} \right) \end{aligned}$$

Figure 3.9: addChildGroup

(REMOVE-FEATURE)

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

Figure 3.10: The REMOVE-FEATURE SOS rule

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 .

3.3.4 Remove group rule

The REMOVE-GROUP rule in Figure 3.18 on page 27 describes the semantics of removing a group in an interval-based feature model. The temporal scope is identified as the existence interval containing the time point for removal. In that interval, the group may not have any children, and there cannot be plans to change the type or move the group within the interval. We check the latter by looking up the type and parent feature during the interval; if the set contains only one type/parent feature then the type and parent feature do not change.

We use the `clampInterval` (Figure 3.11 on the following page), `clampIntervalValue` (Figure 3.12 on the next page), and `clampGroup` (Fig-

```

clampInterval(MAP, tc)
= let { [tstart, tend] } ← MAP [tc]≤
    {v} ← MAP [tc]
    MAP' ← MAP \ [tstart, tend)
    in MAP' [[tstart, tc]] ← v

```

Figure 3.11: clampInterval

```

clampIntervalValue(MAP, tc, v)
= let { [tstart, tend] } ← MAP [tc]≤v
    MAP' ← MAP \v [tstart, tend)
    in MAP' [[tstart, tc]] ←∪ v

```

Figure 3.12: clampIntervalValue

```

clampSetInterval(IS, tc)
= let { [tstart, tend] } ← IS [tc]≤
    IS' ← IS \ [tstart, tend)
    in IS' ∪ { [tstart, tc] }

```

Figure 3.13: clampIntervalSet

```

clampFeature((Fe, Fn, Ft, Fp, Fc), tc)
= (clampSetInterval(Fe, tc)
  , clampInterval(Fn, tc)
  , clampInterval(Ft, tc)
  , clampInterval(Fp, tc)
  , Fc)

```

Figure 3.14: clampFeature

```

clampGroup((Ge, Gt, Gp, Gc), tc)
= (clampSetInterval(Ge)
  , clampInterval(Gt, tc)
  , clampInterval(Gp, tc)
  , Gc)

```

Figure 3.15: clampGroup

```

removeFeatureAt((Ge, Gt, Gp, Gc), featureID, tc)
= (Ge, Gt, Gp
  , clampIntervalValue(Gc, tc, featureID))

```

Figure 3.16: removeFeatureAt

```

removeGroupAt((Fe, Fn, Ft, Fp, Fc), groupID, tc)
= (Fe, Fn, Ft, Fp
  , clampIntervalValue(Fc, tc, groupID))

```

Figure 3.17: removeGroupAt

ure 3.15 on the preceding page) helper functions to update the interval-based feature model. The `clampInterval` function takes an interval map with non-overlapping keys and a time point t_c , and updates the interval key containing t_c to end at t_c . `clampIntervalValue` does the same, but for interval maps with overlapping keys and set values. It takes an interval map, a time point t_c , and a value v , and shortens the interval key containing t_c and v to end at t_c . `clampSetInterval` takes an interval set with non-overlapping values and a time point t_c , and shortens the interval containing t_c .

(REMOVE-GROUP)

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

Figure 3.18: The **REMOVE-GROUP** SOS rule

3.3.5 Algorithm for detecting cycles resulting from move operations

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

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

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

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

Repeat this step until the algorithm terminates:

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

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

```

ancestors((NAMES, FEATURES, GROUPS), featureID, t_n)
= let (F_e, F_n, F_t, F_p, F_c) ← FEATURES[featureID]
    parentGroup ← F_p[t_n]
in
    case parentGroup of
    { parentGroupID } →
        parentGroupID : ancestors((NAMES, FEATURES, GROUPS), parentGroupID, t_n)
    ∅ → []
ancestors((NAMES, FEATURES, GROUPS), groupID, t_n)
= let (G_e, G_t, G_p, G_c) ← GROUPS[groupID]
    { parentFeatureID } ← G_p[t_n]
in
    parentFeatureID : ancestors((NAMES, FEATURES, groups), parentFeatureID, t_n)

```

Figure 3.19: ancestors

3.3.6 Move feature rule

See Figure 3.20 on the next page for the semantics of the **moveFeature** operation. The premise $\neg \text{createsCycle}$ refers to the algorithm described in Section 3.3.5 on the preceding page. The algorithm is not described as a function here, as it is easier to understand written in natural language. An implementation of it can be found in the **TODO: appendix**.

The rule identifies the scope $[t_n, t_{p_2})$ by looking up the interval in F_p containing t_n , and looks up the ID of the original parent group

$F_p [[t_n, t_{p_2}]] = \{\text{oldParentID}\}$. The complex formula checks that the types of the feature and its new parent group are compatible at all times during the temporal scope.

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

(MOVE-FEATURE)

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

Figure 3.20: The **MOVE-FEATURE** SOS rule

3.3.7 Move group rule

See Figure 3.21 on the next page for the semantics of the **moveGroup** operation. The semantics is similar to the one for the **moveFeature** operation, but it differs in that it does not have a check for types. This is because there can only be a conflict between a parent group and a child feature, not a parent feature and a child group. Since only the latter relation changes in this rule, it is not necessary to check that the types are compatible.

TODO: This is hard

(MOVE-GROUP)

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

Figure 3.21: The **MOVE-GROUP** SOS rule

3.3.8 Change feature variation type rule

The rule in Figure 3.22 on the following page shows the semantics of changing the feature variation type of the feature with ID `featureID` at time t_n . The first expression above the line ($F_t[t_n]_{\leq} = \{[t_{t_1}, t_{t_2}]\}$) identifies the upper bound of the temporal scope, t_{t_2} . This is when the feature type was originally planned to change. The next line may be hard to read, but its intent is easier to understand. It checks that all the types a parent group has *while it is the parent of the feature* has a type which is compatible with the new type of the feature. If everything above the line is true, then the **FEATURES** map is updated at `featureID` by shortening the interval key for the original type at t_n , and assigning the new type to the affected interval $[t_n, t_{t_2}]$.

TODO: make it more readable

3.3.9 Change group variation type rule

The rule in Figure 3.24 on the next page is similar to the **changeFeatureVariationType** rule in Figure 3.22 on the following page, and shows the semantics of changing the type of a group. In a similar way to the aforementioned **changeFeatureVariationType** rule, it verifies that the types of all the child groups during the affected interval are compatible with the new group type.

(CHANGE-FEATURE-VARIATION-TYPE)

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

Figure 3.22: The **CHANGE-FEATURE-VARIATION-TYPE** SOS rule

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

Figure 3.23: getTypes

(CHANGE-GROUP-VARIATION-TYPE)

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

Figure 3.24: The **CHANGE-GROUP-VARIATION-TYPE** SOS rule

(CHANGE-FEATURE-NAME)

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

Figure 3.25: The **CHANGE-FEATURE-NAME** SOS rule

3.3.10 Change feature name

The semantics of changing the name of a feature are shown in the **CHANGE-FEATURE-NAME** rule in Figure 3.25. The old name and the next planned name change are identified on the first line ($F_n[t_n] = \{\text{oldName}\}$ 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 4

Soundness

TODO: Move to appendix, but keep a few (maybe 2) examples to show how they work TODO: Prepare the reader for what is to come. To not overwhelm the reader with proofs, the details can be found in the appendix. Here I will emphasize how I prove things, give concrete samples, and point to where the rest of it can be found.

We use an inductive proof structure to prove soundness and modularity for the rule system (Section 3.3 on page 22). The base case is a proof by construction, in which we define a sound interval-based feature model. Using the induction hypothesis that the initial model is sound, we prove that each rule preserves soundness and operates within the previously defined scope (Section 3.2 on page 19).

4.1 Soundness for interval-based feature models

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

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

According to [3], the structural requirements (well-formedness rules) are

- WF1** A feature model has exactly one root feature.
- WF2** The root feature must be mandatory.
- WF3** Each feature has exactly one unique name, variation type and (potentially empty) collection of subgroups.
- WF4** Features are organized in groups that have exactly one variation type.
- WF5** Each feature, except for the root feature, must be part of exactly one group.
- WF6** Each group must have exactly one parent feature.
- WF7** Groups with types ALTERNATIVE or OR must not contain MANDATORY features.

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

These requirements can be translated into rules for interval-based feature models (NAMES, FEATURES, GROUPS). We assume that the first time point in the plan is t_0 .

- IBFM1** An interval-based feature model has exactly one root feature. We assume that the root ID is `rootID`, and that $\text{FEATURES}[\text{rootID}] = (R_e, R_n, R_t, R_p, R_c)$. This also means that $R_e = \{[t_0, \infty)\}$ — the root always exists, and that $R_p = \emptyset$ — the root never has a parent group.
- IBFM2** The root feature must be mandatory. This means that

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

where R_t is the types map of the root feature.

- IBFM3** At any time $t_n \geq t_0$, each feature has exactly one unique name, variation type and (potentially empty) collection of subgroups. Given a feature ID `featureID`, this means that if $\text{FEATURES}[\text{featureID}] = (F_e, F_n, F_t, F_p, F_c)$ and $t_n \in F_e$, then
 - (i) $F_n[t_n] = \{\text{name}\}$ — the feature has exactly one name,
 - (ii) $\text{NAMES}[\text{name}][t_n] = \{\text{featureID}\}$ — the name is unique at the time point t_n ,
 - (iii) $F_t[t_n] = \{\text{type}\}$ with $\text{type} \in \{\text{MANDATORY}, \text{OPTIONAL}\}$ — 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 $\text{groupID} \in \bigcup C$ and $\text{GROUPS}[\text{groupID}] = (G_e, G_t, G_p, G_c)$, then

$G_p[t_n] = \{\text{featureID}\}$ — if a group is listed as a subgroup of a feature, then the feature is listed as the parent of the group at the same time.

- IBFM4** At any time $t_n \geq t_0$, each group has exactly one variation type. Given a group ID groupID , this means that if $\text{GROUPS}[\text{groupID}] = (G_e, G_t, G_p, G_c)$ and $t_n \in \leq G_e$, then $G_t[t_n] = \{\text{type}\}$ for $\text{type} \in \{\text{AND}, \text{OR}, \text{ALTERNATIVE}\}$.
- 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 $\text{featureID} \neq \text{rootID}$, if $\text{FEATURES}[\text{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, and $t_n \in \leq F_e$, then $F_p[t_n] = \{\text{groupID}\}$ with $\text{GROUPS}[\text{groupID}] = (G_e, G_t, G_p, G_c)$, $t_n \in \leq G_e$, and $\text{featureID} \in \bigcup G_c[t_n]$. Conversely, if $\text{featureID} \in \bigcup G_c[t_n]$, then $F_p[t_n] = \text{groupID}$.
- IBFM6** At any time $t_n \geq t_0$, each group must have exactly one parent feature. Formally, given a group ID groupID , if $\text{GROUPS}[\text{groupID}] = (G_e, G_t, G_p, G_c)$ and $t_n \in \leq G_e$, then $G_p[t_n] = \{\text{featureID}\}$, and $\text{FEATURES}[\text{featureID}] = (F_e, F_n, F_t, F_p, F_c)$ with $\text{groupID} \in \bigcup F_c[t_n]$.
- 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 $\text{GROUPS}[\text{groupID}] = (G_e, G_t, G_p, G_c)$, if $F_t[t_n] = \{\text{type}\}$ with $\text{type} \in \{\text{ALTERNATIVE}, \text{OR}\}$, and if $\text{featureID} \in \bigcup F_c[t_n]$ and $\text{FEATURES}[\text{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, then $F_t[t_n] = \{\text{OPTIONAL}\}$.

We must add two additional requirements:

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

4.2 Soundness of each rule

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

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

- (i) proving that the rule operates strictly within the previously defined temporal and spatial scopes (Section 3.2 on page 19),
- (ii) that the rule preserves well-formedness, as defined in the above requirements *IBFM1-9*, and
- (iii) that the rule updates the model correctly, preserving soundness as well as respecting the semantics of the operation.

TODO: Summarize before moving to next section

4.2.1 Soundness of the Add feature rule

See Figure 3.3 on page 23 for the **ADD-FEATURE** rule. Let

addFeature(featureID, name, type, parentGroupID) at $[t_n, t_m) \triangleright$
(NAMES, FEATURES, GROUPS)

be the initial state. Recall that this operation adds the feature with ID featureID to the interval-based feature model (NAMES, FEATURES, GROUPS) from t_n to t_m . We assume that (NAMES, FEATURES, GROUPS) is well-formed, as defined in *IBFM1-9*.

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

Scope Recall from Section 3.2 on page 19 that the temporal scope of this operation is $[t_n, t_m)$, and the spatial scope is the feature itself, the parent group and the name.

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

Lemma 4.1. The **ADD-FEATURE** rule operates strictly within the scope of the **addFeature** operation.

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

Since the rule checks that the feature does not already exist during the temporal scope, it is impossible that $\text{featureID} = \text{rootID}$. Thus the rule does not affect the root feature, and **IBFM1** and **IBFM2** hold for the updated interval-based feature model.

Because we assume that **IBFM8** holds for the original model, and the feature does not exist during $[t_n, t_m)$, the feature has no name, type, or subgroups in the original plan. When we add the feature to the feature model using **setFeatureAttributes**, we give the feature exactly one name and one type during the temporal scope, and the set of child groups is empty. The temporal scope is also added to the feature's existence set, so only the new feature has the ID featureID during the temporal scope. To link the feature ID to the name, the rule sets the feature ID as the value at key **name** in the **NAMES** map during the temporal scope, so the name is unique during the temporal scope. Consequently, **IBFM3** holds.

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

Similarly to the argument for **IBFM3**, the parent group ID is uniquely defined for the feature in **setFeatureAttributes**, and featureID is added to the parent group's set of child features, so the new feature is part of exactly one group. Since we do not remove any other feature IDs from the parent group's set of features, and as we already established that the new feature is not the root feature, **IBFM5** is preserved.

The new feature does not have any subgroups during the temporal scope, and we do not modify the parent group's parent feature. Under the assumption that **IBFM6** holds in the original model, it still holds after applying the **ADD-FEATURE** rule.

The rule verifies that all of the parent group's types are compatible with the added feature's type during the temporal scope, so **IBFM7** holds after applying the rule.

Since the rule adds the temporal scope to the new feature's existence table, and since the parent group exists in the original plan, **IBFM8** is preserved after the rule is applied.

It is furthermore impossible that adding this feature creates a cycle in the modified model. The new feature has no subgroups, so it cannot be part of a cycle. Because of the assumption that **IBFM9** holds in the original

plan, and applying the rule does not introduce a cycle, this requirement still holds.

As the rule operates within the scope (Lemma 4.1 on the previous page), it does not affect any other part of the plan.

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

Lemma 4.2. The **ADD-FEATURE** rule preserves well-formedness of the interval-based feature model.

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

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

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

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

, then due to the semantics of map assignment (Definition 3.2 on page 13), and lookup in interval maps (Definition 3.4 on page 14),

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

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

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

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

According to the semantics of assignment (Section 3.1.2 on page 13) and `setFeatureAttributes` (Figure 3.5 on page 23), and given that

$\text{FEATURES}[\text{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, then

- | | | |
|--|---|-----|
| $[t_n, t_m) \in F_e$ | the feature exists | (1) |
| $F_n[[t_n, t_m)] = \{\text{name}\}$ | the feature has the expected name | (2) |
| $F_t[[t_n, t_m)] = \{\text{type}\}$ | the feature has the expected type | (3) |
| $F_p[[t_n, t_m)] = \{\text{parentGroupID}\}$ | the feature has the expected parent group | (4) |
| $F_c[[t_n, t_m)] = \emptyset$ | the feature has subgroups | (5) |

Statement (1) holds due to the line $F_e \cup [t_n, t_m)$ in `setFeatureAttributes`. The next four hold due to both premises in the rule and modifications in the function. Due to the premise $[t_n, t_m) \not\subseteq F_e$, 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 ($F_n[[t_{start}, t_{end})] \leftarrow \text{name}$), and so is the type ($F_t[[t_{start}, t_{end})] \leftarrow \text{type}$) and the parent group ($F_p[[t_{start}, t_{end})] \leftarrow \text{parentGroupID}$). The child groups are not modified, and so (5) holds.

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

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

, meaning that the feature is in the parent group's set of child features in the updated model. This holds because $\text{GROUPS}[\text{parentGroupID}]$ is assigned `addChildFeature(GROUPS[parentGroupID], [t_n, t_m), featureID)` (see Figure 3.6 on page 24), which modifies G_c by adding `featureID` to the set of child features at interval key $[t_n, t_m)$.

Lemma 4.3. The **ADD-FEATURE** rule updates the interval-based feature model according to the semantics of the **addFeature** operation.

4.2.2 Soundness for the Add group rule

See Figure 3.7 on page 24 for the **ADD-GROUP** rule. Let

$$\begin{aligned} &\text{addGroup}(\text{groupID}, \text{type}, \text{parentFeatureID}) \text{ at } [t_n, t_m) \triangleright \\ &(\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \end{aligned}$$

be the initial state. 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 .

Scope Recall from Section 3.2 on page 19 that the temporal scope of this operation is $[t_n, t_m)$, and the spatial scope is the group itself and the parent feature.

In the premise of the rule, only `groupID` and `parentFeatureID` are looked up in the interval-based feature model. Consequently, the premise stays within the spatial scope of the rule. In the conclusion of the rule, `FEATURES` are assigned to and looked up at `parentFeatureID`, and `GROUPS` at `groupID`. The helper functions `addChildGroup` (Figure 3.9 on page 25) and `setGroupAttributes` (Figure 3.8 on page 24) 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 4.4. The **ADD-GROUP** rule operates strictly within the scope of the **addGroup** operation.

Preserving well-formedness If the **ADD-GROUP** rule is applied, the resulting interval-based feature model must be well-formed according to the well-formedness rules **IBFM1–9**.

The rule does not change the root feature’s existence or type, so it does not violate **IBFM1** or **IBFM2**. The `NAMES` map is left unchanged, and the only change made to a feature is to the parent feature, adding `groupID` to the set of child groups at $[t_n, t_m)$. The only feature modified is the parent feature, and only in its child groups map F_c . Since `parentFeatureID` is assigned to the group’s parent feature table F_p at the same key $[t_n, t_m)$, **IBFM3** holds.

Given that **IBFM8** holds in the original model, and as the rule premise makes certain that the group does not already exist during the interval $[t_n, t_m)$, the group does not have any types, parent features, or child features during the interval. When the rule is applied, the group is given exactly one type and parent feature, and $[t_n, t_m)$ is added to its existence set. Thus **IBFM4**, **IBFM6**, and **IBFM8** hold.

As for **IBFM5**, this requirement holds trivially given that it holds in the original model. No feature is added or removed from any group in the **ADD-GROUP** rule, so this condition is not affected and thus still holds.

Similarly, **IBFM7** will hold in the altered model given that it holds in the original one, since the new group does not contain any features during the temporal scope. For the same reason, the rule does not create a cycle, and so **IBFM9** is true for the altered model.

Lemma 4.5. 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

`setGroupAttributes(GROUPS[groupID], type, parentFeatureID)`

to `GROUPS[groupID]`.

Looking up the added group's ID in the modified model should return the information we put in the operation, and given `GROUPS[groupID] = (Ge, Gt, Gp, Gc)`, the following statements hold:

- | | | |
|--|---|-----|
| $[t_n, t_m) \in G_e$ | the group exists | (1) |
| $G_t[[t_n, t_m)] = \{\text{type}\}$ | the group has the expected type | (2) |
| $G_p[[t_n, t_m)] = \{\text{parentGroupID}\}$ | the group has the expected parent feature | (3) |
| $G_c[[t_n, t_m)] = \emptyset$ | the group has no children | (4) |

Statement (1) holds due to the line $G_e \cup [t_n, t_m)$ in `setGroupAttributes` (Figure 3.8 on page 24). 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 `FEATURES[parentFeatureID] = (Fe, Fn, Ft, Fp, Fc)`, then

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

In the **ADD-GROUP** rule,

`addChildGroup(FEATURES[parentFeatureID], [tn, tm), groupID)`

is assigned to `FEATURES[parentFeatureID]`. The function `addChildGroup` (Figure 3.9 on page 25) adds `groupID` to the set of child features at key $[t_n, t_m)$, so according to the semantics of $\overset{\cup}{\leftarrow}$, it is indeed true that the group is in the parent feature's set of child group in the temporal scope.

Lemma 4.6. The **ADD-GROUP** rule updates the interval-based feature model according to the semantics of the **addGroup** operation.

4.2.3 Soundness of the Remove feature rule

See Figure 3.10 on page 25 for the **REMOVE-FEATURE** rule. Let

$$\begin{aligned} &\mathbf{removeFeature}(\text{featureID}) \text{ at } t_n \triangleright \\ &(\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \end{aligned}$$

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

Scope As defined in Section 3.2 on page 19, the **removeFeature** operation's temporal scope is $[t_n, t_k)$, where t_k is the time point in which the feature was originally planned to be removed. We can see from the description above that $t_k = t_j$; the end point in the feature's existence set containing t_n . We then have that the scope is defined as $[t_n, t_j)$. In the rule, we find the interval $[t_i, t_j)$ by looking up

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

According to the semantics of $\text{IS}[t_n]_{\leq}$, it is true then that $[t_i, t_j) = [t_{e_1}, t_{e_2})$, and so the temporal scope of the rule is $[t_n, t_{e_2}) = [t_n, t_j)$. Clearly, all time points looked up in the premise of the rule are contained within this interval, but the conclusion requires further examination. The **NAMES** map is assigned $\text{clampInterval}(\text{NAMES}[\text{name}], t_n)$ at key `name`. In clampInterval (Figure 3.11 on page 26), the interval $[t_{n_1}, t_{n_2})$ containing t_n in $\text{NAMES}[\text{name}]$ 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

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

. In clampFeature (Figure 3.14 on page 26), 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

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

at key `parentGroupID`. This helper function (Figure 3.16 on page 26) modifies the parent group's set of subfeatures by calling

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

, which behaves similarly to `clampInterval` by clamping the interval containing t_n . The difference is that it removes only `featureID` from the set of subfeatures, and adds the feature to the set of subfeatures at the shortened interval. We conclude that this modification, too, happens within the temporal scope of the operation, as looking up any time point outside of the temporal scope will return the same results as the original plan.

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

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

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

Lemma 4.7. The **REMOVE-FEATURE** rule operates strictly within the scope of the **removeFeature** operation.

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

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

, 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, $t_n \notin F_e$ due to the semantics of `clampFeature` (Figure 3.14 on page 26), so **IBFM3** holds trivially. The only change made to a group is by the function `removeFeatureAt` (Figure 3.16 on page 26), which removes the feature

from the parent group's map of subgroups during $[t_n, t_j)$. Hence **IBFM5** holds, and since that function does not modify the types map G_t of the group, **IBFM4** holds given that it is true for the original model.

The premise $F_c [[t_n, t_{e_2}]] = \emptyset$ ensures that the feature to be removed does not have any subgroups during the temporal scope, so no group is left without a parent in the updated model. Thus **IBFM6** holds.

Suppose that the parent group has the type **ALTERNATIVE** or **OR** at some point during the temporal scope. In the original model, no child feature of the group has type **MANDATORY** due to the assumption that **IBFM7** is true. The **REMOVE-FEATURE** rule does not add any features or change a feature type, so this requirement still holds for the modified model.

After applying the rule, we have that $[t_n, t_j) \not\subseteq_{\approx} F_e$, which means that the feature does not exist during the temporal scope of the operation. To fulfil **IBFM8**, we must furthermore have that $F_n [[t_n, t_j]] = F_t [[t_n, t_j]] = F_p [[t_n, t_j]] = F_c [[t_n, t_j]] = \emptyset$, and that $\text{featureID} \notin \text{NAMES}[\text{name}] [[t_n, t_j]]$. 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. $F_c [[t_n, t_j]] = \emptyset$ is a premise in the rule (since $t_j = t_{e_2}$). As for the **NAMES** map, the mapping from name to $[t_i, t_j) \mapsto \text{featureID}$ is replaced by $[t_i, t_n) \mapsto \text{featureID}$ in the function $\text{clampInterval}(\text{NAMES}[\text{name}], t_n)$. Hence **IBFM8** is true for the altered interval-based feature model.

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

Lemma 4.8. 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 it was originally planned to be removed. Then if $\text{FEATURES}[\text{featureID}] = (F_e, F_n, F_t, F_p, F_c)$, and $F_e[t_n]_{\leq} = [t_i, t_j)$, then

- | | | |
|--|---------------------------------|-----|
| $[t_n, t_m) \not\subseteq_{\approx} F_e$ | the feature does not exist | (1) |
| $F_n [[t_n, t_m]] = \emptyset$ | the feature has no name | (2) |
| $F_t [[t_n, t_m]] = \emptyset$ | the feature has no type | (3) |
| $F_p [[t_n, t_m]] = \emptyset$ | the feature has no parent group | (4) |
| $F_c [[t_n, t_m]] = \emptyset$ | the feature has no subgroups | (5) |

Since we established $[t_n, t_m) \notin_{\approx} F_e$ in the above paragraph, these statements follow directly from Lemma 4.8 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 sub-feature.

Lemma 4.9. The **REMOVE-FEATURE** rule updates the interval-based feature model according to the semantics of the **removeFeature** operation.

4.2.4 Soundness of the Remove group rule

This proof is analogous to the one for the **REMOVE-FEATURE** rule. See Figure 3.18 on page 27 for the **REMOVE-GROUP** rule. Let

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

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 $\text{GROUPS}[\text{groupID}] = (G_e, G_t, G_p, G_c)$, and $[t_i, t_j) \in G_e$, with $t_i \leq t_n < t_j$. This means that the original plan added the group at time t_i and removed it at t_j , with the possibility that $t_j = \infty$. In the latter case, there was no plan to remove the group originally.

Scope todo

TODO: Ask if it is necessary to include this paragraph or if it is enough to say that this is very similar to the Scope section of Remove-Feature

As defined in Section 3.2 on page 19, 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 $\text{IS}[t_n]_{\leq}$, it is true then that $[t_i, t_j) = [t_{e_1}, t_{e_2})$, and so the temporal scope of the rule is $[t_n, t_{e_2}) = [t_n, t_j)$. Clearly, all time points looked up in the premise of the rule are contained within this

interval, but the conclusion requires further examination. The NAMES map is untouched and thus outside the scope.

The GROUPS map is modified at key groupID by assigning

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

. In `clampGroup` (Figure 3.15 on page 26), 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 `clampInterval(MAP, t_c)` clamps the mapping with an interval key containing t_c to end at t_c . Due to *IBFM8*, it is impossible that the group has a type or parent feature after t_j , which is the time point when the group was originally planned to be removed. Furthermore, the premise of the rule requires that $G_t[[t_n, t_j]] = \{\text{type}\}$ and $G_p[[t_n, t_j]] = \{\text{parentFeatureID}\}$, 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

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

at key `parentFeatureID`. This helper function (Figure 3.17 on page 26) modifies the parent feature's set of subgroups by calling

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

, which behaves similarly to `clampInterval` by clamping the interval containing t_n . The difference is that it removes only `groupID` from the set of child groups, and adds the group to the set of subgroups at the shortened interval. We conclude that this modification, too, happens within the temporal scope of the operation, as looking up any time point outside of the temporal scope will return the same results as the original plan.

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

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

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

Lemma 4.10. The **REMOVE-GROUP** rule operates strictly within the scope of the **removeGroup** operation.

Preserving well-formedness Let

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

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

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

Since a premise of the rule is

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

, the group does not have any subfeatures during the temporal scope in the original interval-based feature model. Due to the assumption that **IBFM5** is true for the original model, no feature has `groupID` listed as its parent group, so no feature is left without a parent group when the group is removed from the temporal scope. It follows that **IBFM5** holds for the updated interval-based feature model as well.

As previously mentioned, $[t_n, t_j] \not\subseteq G'_e$, 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 $G'_t[[t_n, t_j]] = G'_p[[t_n, t_j]] = G'_c[[t_n, t_j]] = \emptyset$. Recalling that $t_{e_2} = t_j$, then by definition of `clampGroup` and the premise $G_t[[t_n, t_{e_2}]] = \{\text{type}\}$ in **ADD-GROUP**, we have that

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

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.

Lemma 4.11. The **REMOVE-GROUP** rule preserves well-formedness of the interval-based feature model.

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

Lemma 4.12. The **REMOVE-GROUP** rule updates the interval-based feature model according to the semantics of the **removeGroup** operation.

4.2.5 Soundness of the Move feature rule

See Figure 3.20 on page 29 for the **MOVE-FEATURE** rule. Let

$$\begin{aligned} &\mathbf{moveFeature}(\text{featureID}, \text{newParentID}) \text{ at } t_n \triangleright \\ &(\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \end{aligned}$$

be the initial state, and

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

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

Scope Recall that the temporal scope of the move-feature rule is $[t_n, t_k)$ (Section 3.2 on page 19), 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 $[t_{p_1}, t_{p_2}) \mapsto \text{parentGroupID}$ 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 3.3.5 on page 27, since this is also referenced in the rule by `¬createsCycle`. Here, t_{p_2} is called t_e , and the algorithm states that it only looks at time points between t_n and t_e . Thus the rule operates strictly within the temporal scope of the **moveFeature** operation.

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

Lemma 4.13. The **MOVE-FEATURE** rule operates strictly within the scope of the **moveFeature** operation.

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

The modification made to `FEATURES[featureID]` is to the parent group map F_p by

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

As discussed in earlier sections (e.g. Section 4.2.3 on page 42), `clampInterval` replaces a mapping $[[t_i, t_j) \mapsto v]$ by $[[t_i, t_n) \mapsto v]$, with $t_n \leq t_j$. Thus `clampInterval(F_p, t_n)` has no parent group during the temporal scope. The subsequent assignment of `newParentID` to $[t_n, t_{p_2})$ ensures that the feature has exactly one parent group during the temporal

scope. This relation is reflected in the GROUPS' map, with

$\text{GROUPS}' =$

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

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

By the premise

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

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.

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

The intention of the cycle detection algorithm in Section 3.3.5 on page 27 is to uphold **IBFM9**. Given the assumption that the original interval-based feature model contains no cycles, if the altered model contains a cycle then the **moveFeature** operation introduced it, and the feature being moved must be part of the cycle. This could only happen if the feature became part of its own subtree during the temporal scope, which means that at some point, the feature occurs in its own list of ancestors. The algorithm looks at the feature's *new* ancestors, meaning the ancestors that the feature does not have in the original plan, but does in the new one. It then checks that none of those ancestors are moved to the feature's subtree. Thus the rule preserves **IBFM9**.

Lemma 4.14. The **MOVE-FEATURE** rule preserves well-formedness of the interval-based feature model.

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

- (i) The feature's parent group should be `newParentID` during the temporal scope
- (ii) The feature should not appear in the original parent group's set of subfeatures during the temporal scope
- (iii) The feature should appear in the new parent group's set of subfeatures

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

$$F'_p = \text{clampInterval}(F_p, t_n) \llbracket [t_n, t_{p_2}) \rrbracket \leftarrow \text{newParentID}$$

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

By Lemma 4.14 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 subfeatures during the temporal scope, and is in the new parent group's set of subfeatures.

Lemma 4.15. The **MOVE-FEATURE** rule updates the interval-based feature model according to the semantics of the **moveFeature** operation.

4.2.6 Soundness of the Move group rule

See Figure 3.21 on page 30 for the **MOVE-GROUP** rule. Let

$$\text{moveGroup}(\text{groupID}, \text{newParentID}) \text{ at } t_n \triangleright \\ (\text{NAMES}, \text{FEATURES}, \text{GROUPS})$$

be the initial state, and

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

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

Scope Recall that the temporal scope of the **MOVE-GROUP** rule is $[t_n, t_k]$ (Section 3.2 on page 19), 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 $[t_{p_1}, t_{p_2}) \mapsto \text{parentFeatureID}]$ 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 3.3.5 on page 27, since this is also referenced in the rule by $\neg \text{createsCycle}$. Here, t_{p_2} is called t_e , and the algorithm states that it only looks at time points between t_n and t_e . Thus the rule operates strictly within the temporal scope of the **moveGroup** operation.

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

Lemma 4.16. The **MOVE-GROUP** rule operates strictly within the scope of the **moveGroup** operation.

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

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

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

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

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

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

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

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

The intention of the cycle detection algorithm in Section 3.3.5 on page 27 is to uphold **IBFM9**. Given the assumption that the original interval-based feature model contains no cycles, if the altered model contains a cycle then the **moveGroup** operation introduced it, and the group being moved must be part of the cycle. This could only happen if the group became part of its own subtree during the temporal scope, which means that at some point, the group occurs in its own list of ancestors. The algorithm looks at the group's *new* ancestors, meaning the ancestors that the group does not have in the original plan, but does in the new one. It then checks that none of those ancestors are moved to the group's subtree. Thus the rule preserves **IBFM9**.

Lemma 4.17. The **MOVE-GROUP** rule preserves well-formedness of the interval-based feature model.

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

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

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

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

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

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

By Lemma 4.17 on the previous page and **IBFM5**, (ii) and (iii) follow from (i). In other words, since the updated interval-based feature model is well-formed, and the group's parent feature during the temporal scope is `newParentID`, the group is not in the original parent feature's set of subgroups during the temporal scope, and is in the new parent feature's set of subgroups.

Lemma 4.18. The **MOVE-GROUP** rule updates the interval-based feature model according to the semantics of the **moveGroup** operation.

4.2.7 Soundness of the Change feature variation type rule

See Figure 3.22 on page 31 for the **CHANGE-FEATURE-VARIATION-TYPE** rule. Let

$$\begin{aligned} &\text{changeFeatureVariationType}(\text{featureID}, \text{type}) \text{ at } t_n \triangleright \\ &(\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \end{aligned}$$

be the initial state, and

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

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

Scope Recall that the temporal scope of the **CHANGE-FEATURE-VARIATION-TYPE** rule is $[t_n, t_k)$ (Section 3.2 on page 19), 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 $[t_{t_1}, t_{t_2}] \mapsto \text{oldType}$ 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

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

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

Lemma 4.19. The **CHANGE-FEATURE-VARIATION-TYPE** rule operates strictly within the scope of the **changeFeatureVariationType** operation.

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

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

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

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

looks up all parent mappings overlapping the temporal scope ($[t_{p_1}, t_{p_2}] \mapsto p$), finds the types each parent group has during the scope and *while* it is

the parent of the feature, and verifies that those types are compatible. Thus *IBFM7* is preserved.

Lemma 4.20. 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 $\text{FEATURES}'[\text{featureID}] = (F'_e, F'_n, F'_t, F'_p, F'_c)$ 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 $t_n \leq t_i < t_{t_2}$,

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

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

Lemma 4.21. The **CHANGE-FEATURE-VARIATION-TYPE** rule updates the interval-based feature model according to the semantics of the **changeFeatureVariationType** operation.

4.2.8 Soundness of the Change group variation type rule

See Figure 3.24 on page 31 for the **CHANGE-GROUP-VARIATION-TYPE** rule. Let

$$\begin{aligned} &\text{changeGroupVariationType}(\text{groupID}, \text{type}) \text{ at } t_n \triangleright \\ &(\text{NAMES}, \text{FEATURES}, \text{GROUPS}) \end{aligned}$$

be the initial state, and

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

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

Scope Recall that the temporal scope of the **CHANGE-GROUP-VARIATION-TYPE** rule is $[t_n, t_k)$ (Section 3.2 on page 19), 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 $[[t_{t_1}, t_{t_2}] \mapsto \text{oldType}]$ in G_t , with oldType being the type of the group at time t_n , and this stops being the case at t_{t_2} . Thus the temporal scope of this operation is $[t_n, t_{t_2})$. The only interval looked up or assigned to in the rule is $[t_n, t_{t_2})$, so the rule operates strictly within the temporal scope of the operation.

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

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

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

Lemma 4.22. The **CHANGE-GROUP-VARIATION-TYPE** rule operates strictly within the scope of the **changeGroupVariationType** operation.

Preserving well-formedness The modification to G_t

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

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

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

looks up all subfeature mappings overlapping the temporal scope $([[t_{c_1}, t_{c_2}) \mapsto \{f_1, f_2, \dots\}])$, finds the types each subfeature has during the scope and *while* it is the subfeature of the group, and verifies that those types are compatible. Thus **IBFM7** is preserved. As no changes are made

to any other part of the interval-based feature model, the other requirements *IBFM1–3*, *IBFM5–6*, and *IBFM8–9* hold trivially.

Lemma 4.23. 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 $\text{GROUPS}'[\text{groupID}] = (G'_e, G'_t, G'_p, G'_c)$ 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 $t_n \leq t_i < t_{t_2}$,

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

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

Lemma 4.24. The **CHANGE-GROUP-VARIATION-TYPE** rule updates the interval-based feature model according to the semantics of the **changeGroupVariationType** operation.

4.2.9 Soundness of the Change feature name rule

See Figure 3.25 on page 32 for the **CHANGE-FEATURE-VARIATION-TYPE** rule. Let

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

be the initial state, and

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

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

Scope Recall that the temporal scope of the **CHANGE-FEATURE-NAME** rule is $[t_n, t_k)$ (Section 3.2 on page 19), 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 $[t_{n_1}, t_{n_2}) \mapsto \text{oldName}$ 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 $\text{FEATURES}[\text{featureID}]$, and the only names looked up or assigned to are oldName and name . The GROUPS map is not modified or looked up in by the rule. Clearly, the rule stays within the spatial scope.

Lemma 4.25. The **CHANGE-FEATURE-NAME** rule operates strictly within the scope of the **changeFeatureName** operation.

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

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

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

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

Here, the interval containing t_n in $\text{NAMES}[\text{oldName}]$ 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.

Lemma 4.26. 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 $\text{FEATURES}'[\text{featureID}] = (F'_e, F'_n, F'_t, F'_p, F'_c)$ has the 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 $t_n \leq t_i < t_{n_2}$,

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

Additionally, we should have $\text{NAMES}'[\text{name}] = \text{featureID}$. 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 4.27. The **CHANGE-FEATURE-NAME** rule updates the interval-based feature model according to the semantics of the **changeFeatureName** operation.

4.3 Soundness of the rule system

Section 4.2 on page 36 shows that each the SOS rules operates within the scope of an operation.¹

Theorem 4.28. The rule system supports local modification and verification of interval-based feature models.

Section 4.2 on page 36 shows that each of the SOS rules preserves well-formedness of the interval-based feature model.²

Theorem 4.29. The rule system for local modification of interval-based feature models is well-formed.

Section 4.2 on page 36 shows that each of the SOS rules behaves according to the semantics of the operation in question.³

Theorem 4.30. The rule system for local modification of interval-based feature models modifies the model correctly.

¹See lemmas 4.1 on page 37, 4.4 on page 40, 4.7 on page 43, 4.10 on page 46, 4.13 on page 49, 4.16 on page 52, 4.19 on page 55, 4.22 on page 57, 4.25 on the previous page

²See lemmas 4.2 on page 38, 4.5 on page 41, 4.8 on page 44, 4.11 on page 48, 4.14 on page 50, 4.17 on page 53, 4.20 on page 56, 4.23 on page 58, 4.26 on the previous page

³See lemmas 4.3 on page 39, 4.6 on page 42, 4.9 on page 45, 4.12 on page 48, 4.15 on page 51, 4.18 on page 54, 4.21 on page 56, 4.24 on page 58, 4.27

Chapter 5

Implementation

TODO: Move to appendix

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

Part III

Conclusion

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

Bibliography

- [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] 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>.