Formalization and model checking of feature model evolution plans

Eirik Halvard Sæther and Ida Sandberg Motzfeldt

Summer 2019

Contents

1 Introduction

Software engineering methodologies for highly-variable software systems lack support for planning the long-term evolution of software. This project will try to aid in tackling some of the problems related to the long term evolution planning of software product lines, by developing a formal methodology to explicitly handle the evolution of a software product line, as well as specify temporal constraints for plan analysis. This plan is then verified by a model checker, to make sure the plan is producing a valid feature model at every (future) point in time. With a simple temporal grammar, the user can express constraints that can hold over the entire planned evolution. This is done by using structural operational semantics to formalize the evolution of the software product line, as well as using linear temporal logic to check temporal constraints. The Maude system is used to implement the specification.

This project is part of a larger collaboration between the University of Oslo and Technische Universität Braunschweig (TUBS), where the goal is to deal with issues in long-term evolution planning by developing and formalizing new methods. The collaboration will continue for two years, expanding on and adding to the work we have done this summer.

The complete Maude implementation can be found at

https://github.com/idamotz/LTEP-summer-project

2 Background

2.1 Software product lines

A software product line (SPL) is a family of closely related software systems. These systems can have several features in common, as well as features specific to one or more systems. They are used to make highly configurable systems, where a final product, called a *variant*, is defined by the combination of features that are chosen.

There exist several SPL engineering methodologies to implement a whole software product line. These methodologies capitalize on the similarities and differences between the various variants. Instead of developing and maintaining several code bases for each variant, these methodologies combine all these code bases by explicitly encoding their similarities and differences. This makes it easier to develop and maintain features across projects. Examples of this are DeltaJava[1], FeatureIDE[2], pure::variants[3], etc.

2.2 Feature models

All the possible variants of a software product line can be defined in terms of a *feature model*. The feature model describes dependencies between features, and how they are related to each other. A feature model is a tree structure, where each vertex represents a feature, and a feature can contain *groups* of more features. See Figure 1 for an example of a (simplified) feature model.

[1]https://deltajava.org/
[2]https://featureide.github.io/
[3]https://www.pure-systems.com/products/pure-variants-9.html

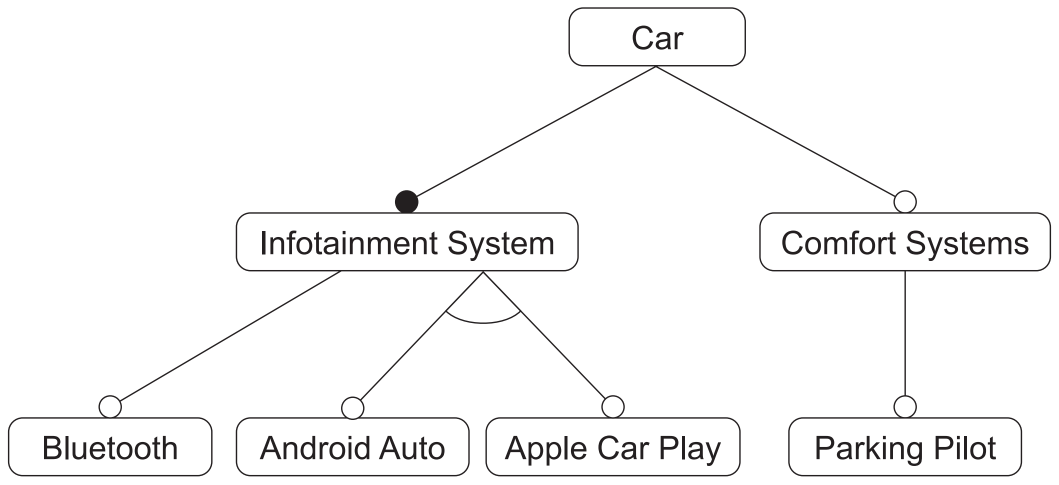


Figure 1: Example feature model

The small black dot above Infotainment system means that the feature is *mandatory*, which means it must be selected for all variants where its parent feature (here Car) is selected. A white dot (as seen above the Bluetooth feature) means that the feature is *optional*, and does not have to be selected in a variant. The white triangle connecting Android Auto and Apple Car Play is a visualization of an alternative group, or XOR group. This means that exactly one of Android Auto and Apple Car Play must be selected. A black triangle (not shown here) is the syntax for the OR group, where at least one feature in the group must be selected in a variant. The AND group is not shown explicitly, but if it is not an XOR or OR group, it is an AND group. An AND group gives no restrictions on which features can be selected.

All features have an associated ID (not shown in example), a name, a (possibly empty) collection of subgroups, and a *feature variation type* (optional or mandatory as explained above). All groups have an associated ID (not shown in example), a *group variation type* (AND, OR or XOR as explained above), and a collection of subfeatures.

There are some rules for feature models:

- The root feature must be mandatory - it must be selected in all variants
- All IDs and names are unique
- If a group has group variation type XOR or OR, it cannot contain any mandatory features.

In addition, it is common to define *cross-tree constraints* when there are dependencies that cannot be visualized in a tree; e.g. Parking Pilot cannot be selected without Bluetooth being selected. For more information on SPLs and feature models, see [4].

2.3 Evolution of feature models

The current tools only deal with the construction of SPLs, and not the evolution. On large software product lines, the planning of long term evolution is lacking, and thus creating a risk of increased development costs.

This project will try to aid in tackling some of the problems related to this long term evolution planning, by developing a formal methodology to explicitly handle the evolution of the SPL, as well as specify temporal constraints for plan analysis. By having a tool for specifying the future of the product line, the team members will have a more common understanding of what they are building. This plan is then checked by a model checker, to make sure the plan is producing a valid feature model at every (future) point in time. With temporal constraints, the user can express constraints that can hold over the entire planned evolution. Examples of this is constraints such as "feature Bluetooth exists while feature Parking Pilot exists".

2.4 Evolution plans

In this project, we model evolution using evolution plans, imagining that we receive plans in the form of an initial feature model, followed by an ordered list of plans associated with time points. An example evolution plan is the initial feature model followed by the plan shown in Figure 2.

Implementing this plan results in the feature model shown in Figure 1. There are several operations on feature models, which makes it possible to add, move or remove features and groups, change group or feature types, etc. Part of our goal is to formalize and implement the feature models and their evolution, so that we can verify that no plans break the basic rules of a feature model. Evolution plans are discussed in more detail in section 3.1.

2.5 Structural operational semantics

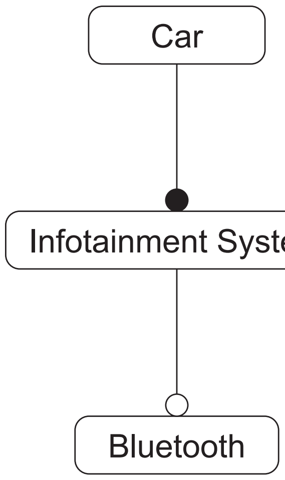
Structural operational semantics (SOS), commonly referred to as small-step semantics, is a form of semantics describing each individual step in the execution of a program. An SOS specification is a set of inference rules, where each rule describes a transition for a piece of valid syntax. A rule has the form

$$\frac{\text{Condition}}{\text{State} \implies \text{State'}}$$

The rule has three parts; the condition(s) above the line, the State (possibly containing variables) that may be part of a program, and the State' which this part will be rewritten to if all the conditions hold. SOS can be used to formally and precisely model the behaviors of a system, and as a basis for formal analysis. We use structural operational semantics to formalize feature models, as well as feature model evolution plans. For more information on structural operational semantics, see [2].

2.6 Rewriting logic

Rewriting logic is a computational logical and semantic framework which is useful for modeling dynamic systems and reason about change in a system. A system can be modeled in a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, where Σ is the syntax, containing terms and state constructors, which are algebraic expressions and variables; E is the equational theory, which specifies the algebraic identities for the



At time 1:

add an XOR group to Infotainment System.
add feature Android Auto to the Infotainment System XOR group
add feature Car Play to the Infotainment System XOR group

At time 2:

add feature Comfort Systems to the Car AND group
add an AND group to Comfort Systems
add feature Parking Pilot to the Comfort Systems AND group

Figure 2: Example evolution plan

states; and R is the collection of *rewrite rules* on the form $t \to t'$, where t and t' are algebraic expressions (often containing variables) in the syntax Σ. These rules model the dynamics of the system, by giving patterns t that, if matched in a subterm of a state, yields another pattern t' which is substituted into the state - in much the same way as the rules in SOS. The rules of rewriting logic are very similar to the rules in structural operational semantics, which is why we have chosen this logical framework for creating our executable model of the evolution plans. For more information on rewriting logic, see [1] and [3].

2.7 Linear temporal logic

Linear temporal logic (LTL) is a logic that is used to express properties referring to time. In LTL, you can write formulas that express what is going to happen in the future; i.e. a condition will always be true, a condition will hold until some other condition holds, etc.

Each behavior is defined from an initial state t_0, with an infinite sequence of one-step sequential rewrites. Since our system does not have infinite computations, A finite computation, $t_0 \longrightarrow t_1 \longrightarrow \ldots \longrightarrow t_n$, is represented in LTL as the infinite sequence (path) $t_0 \longrightarrow t_1 \longrightarrow \ldots \longrightarrow t_n \longrightarrow t_n \longrightarrow$

$\ldots \longrightarrow t_n \longrightarrow \ldots$. Maude's model checker adds this self loop in the deadlocked state automatically.

The syntax of LTL is built up inductively, with *atomic propositions* as the basic building blocks. An atomic proposition is a *state proposition*, which is either true or false in a state. *true* and *false* are also LTL formulas. If both φ and ψ are LTL formulas, then the following are also LTL formulas:

- $\neg\varphi$ (not φ)
- $\varphi \wedge \psi$ (φ and ψ)
- $\varphi \vee \psi$ (φ or ψ)
- $\varphi \rightarrow \psi$ (if φ then ψ)
- $\Box\varphi$ (always φ)
- $\Diamond\varphi$ (eventually φ)
- $\varphi \mathscr{U} \psi$ (φ until ψ, and eventually ψ will be true)
- $\varphi \mathscr{W} \psi$ (φ until ψ, but ψ might never be true)
- $\bigcirc\varphi$ (φ in the next state)

To check if a formula is true, you check from an initial state. The formula is then true if the formula holds for every possible sequence of rewrite steps. An example of a formula is $\Box(\varphi \rightarrow \Diamond\psi)$, which roughly means "If φ happens at any point, it will always be followed by ψ at a later point. We use linear temporal logic to model temporal constraints for feature models. For a more detailed description of the linear temporal logic, see [3]

2.8 Introduction to Maude

To model the evolution of feature models, we use the Maude system. Maude is a formal modeling language used to define, execute and analyze formal specifications. Maude is based on the theory of rewriting logic, which makes it a good match for our purposes. In Maude, terms are defined by *sorts*, which makes it possible to give inductive ways to define the terms. Defining sorts is quite powerful, especially for commutative data structures. You can simply note that a constructor is commutative by adding the *comm* property, which makes defining equations and rules much closer to the formal semantics.

The non-dynamic parts of the system are modeled by equations in Maude. When defining equations, you give a pattern on the left hand side, and what it is equal to on the right hand side. Equations then only apply if the value applied matches the pattern in the equation. The Maude interpreter then evaluates expressions by applying equations "from left to right" until no equation can be applied. We rely heavily on pattern matching in our specification.

To capture the dynamic and changing parts of a system, you specify *rewrite rules*. Rewrite rules can be used to model *non-determinism*; however, this is not relevant for our use case. Instead, we use rewrite rules to capture the changes in our system, so that we later can *model check* the specification using temporal logic. Maude gives a high-performance model checker to check requirements from an initial state and a set of rewrite rules. Following is a simple example of a simple (functional) Maude module.

```
fmod BOOLEAN is
    --- Defining the sort Boolean,
    --- currently no terms related to the sort
    sort Boolean .

    --- Adding two constants to the sort
    --- Note that there is a ctor flag, which means this is
    --- not equations, but rather just a term
    ops true false : -> Boolean [ctor] .

    --- Adding three functions for Booleans.
    --- Underscores are used to define where the arguments are put
    --- The lack of a ctor flag means that these are functions
    --- that are meant to be applied using equations which are defined below
    --- For example: (not true) or false
    op not_ : Boolean -> Boolean [prec 53] .
    op _and_ : Boolean Boolean -> Boolean [prec 55] .
    op _or_ : Boolean Boolean -> Boolean [prec 59] .

    --- Variables are defined explicitly
    var B : Boolean .

    --- If you have a term that matches a left hand side
    --- of any of the following equations, it will be reduced
    --- to the right hand side
    eq not false = true .   eq not true = false     .
    eq true and B = B    .   eq false and B = false .
    eq true or B = true .   eq false or B = B       .
endfm

--- Example usage of the module:
red (not true) or false .
---> (not true) or false
---> false or false
---> false
--- Not more equations to be applied, term on normal form
```

Figure 3: Simple Maude example

Maude's model checker

Maude has a built-in model checker, which can be used to analyze models using LTL formulas, as long as the set of states reachable from the initial state is finite. To simulate an infinite sequence of states as required by LTL, Maude automatically adds a self-loop in the last state of a terminating specification.

To use this model checker, we define a sort s as a subsort of `State` to define which terms should be interpreted as states. Next, we declare the atomic propositions as operators with sort `Prop`, and define their semantics by giving equations on the form `eq ⟨state⟩ |= ⟨prop⟩ .` The formulas that can be tested using the model checker are syntactically similar to LTL. Table 1 shows the Maude symbols and the equivalent LTL symbols.

Maude	**LTL**
`~`	$\neg$
`/\`	$\wedge$
`\/`	$\vee$
`->`	$\rightarrow$
`[]`	$\square$
`<>`	$\lozenge$
`U`	$\mathscr{U}$
`W`	$\mathscr{W}$
`O`	$\bigcirc$

Table 1: Maude's LTL syntax

After implementing such a model checker, we can use it to test whether these formulas actually hold for a specification. This is done by using the built-in function *modelCheck* in the following way: `modelCheck(⟨initialState⟩, ⟨formula⟩)`. If a formula is not satisfied by the system, Maude outputs a counterexample to the formula, in the form of a sequence of states for which the formula does not hold. Otherwise, Maude simply outputs that the formula holds. We use Maude's model checker to implement the temporal constraints in linear temporal logic.

For more information on Maude, see [3].

3 Semantics of feature model evolution plans

3.1 Formalizing feature models and operations

In this section we will give exact mathematical definitions to the terms involved in feature models and operations on feature models. To define and capture exactly how operations like *add feature* is performed on feature models, we will use structural operational semantics.

Defining the terms

A feature model is defined as a term FM(rootID, FT), where rootID is the ID of the root feature, and FT is a *feature table.* A feature table consists of mappings from feature IDs to feature terms, where feature terms are tuples (name, parentFid, $\overline{\text{groups}}$, featureType), and a mapping looks like this:

$$[\text{featureID} \mapsto (\text{name}, \text{parentFid}, \overline{\text{groups}}, \text{featureType})]$$

where featureID is the ID of a feature; name is the name of the feature; parentFid is the feature ID for the parent feature of the feature; $\overline{\text{groups}}$ is a set of groups directly below the feature; and featureType is the feature variation type (optional or mandatory). A group is defined as a tuple (groupID, groupType, $\overline{\text{features}}$), where groupID is the ID of the group; groupType is the group variation type (AND, OR, or XOR); and $\overline{\text{features}}$ is a set of feature IDs. We use + as constructor for the feature table, and it is a commutative and associative operation with identity element ε. We also have a lookup function for the feature table, defined as

$$\begin{aligned}&\mathrm{FT}(\mathrm{featureID}) = (\mathrm{name}, \mathrm{parentFid}, \mathrm{groups}, \mathrm{featureType})\\&\text{if } [\mathrm{featureID} \mapsto (\mathrm{name}, \mathrm{parentFid}, \mathrm{groups}, \mathrm{featureType})] \in \mathrm{FT}\end{aligned}$$

Furthermore, we use :: as constructor for sets, which is also both associative and commutative with identity $\emptyset$.

Example model

Using the initials of the features as IDs and these IDs combined with numbers for their groups, the feature model example shown in Figure 1 can be modeled as the term

$$\begin{aligned}&\mathrm{FM}(\mathrm{C},\\
&\quad [\mathrm{C} \mapsto (\mathrm{Car}, \perp, (\mathrm{C1}, \mathrm{AND}, \mathrm{IS} :: \mathrm{CS}), \mathrm{mandatory})] +\\
&\quad [\mathrm{IS} \mapsto (\text{Infotainment System}, \mathrm{C}, (\mathrm{IS1}, \mathrm{AND}, \mathrm{B}) :: (\mathrm{IS2}, \mathrm{XOR}, \mathrm{AA} :: \mathrm{ACP}), \mathrm{mandatory})] +\\
&\quad [\mathrm{B} \mapsto (\mathrm{Bluetooth}, \mathrm{IS}, \emptyset, \mathrm{optional})] +\\
&\quad [\mathrm{AA} \mapsto (\text{Android Auto}, \mathrm{IS}, \emptyset, \mathrm{optional})] +\\
&\quad [\mathrm{ACP} \mapsto (\text{Apple Car Play}, \mathrm{IS}, \emptyset, \mathrm{optional})] +\\
&\quad [\mathrm{CS} \mapsto (\text{Comfort Systems}, \mathrm{C}, (\mathrm{CS1}, \mathrm{AND}, \mathrm{PP}), \mathrm{optional})] +\\
&\quad [\mathrm{PP} \mapsto (\text{Parking Pilot}, \mathrm{CS}, \emptyset, \mathrm{optional})])\end{aligned}$$

Figure 4: Example feature model term

Formalizing evolution operations

Our goal using structural operational semantics is to formalize the operations on a feature model, as well as the concept of evolution plans. First we will define the terms used to express the evolution operations, then show how the application of one of these operations is expressed using SOS.

As an example, we will show how the *addFeature* operation is performed on a feature model. The SOS rule shown in Figure 5 describes the semantics of this operation.

Below the line we can see that the part before the arrow can be rewritten to the part after the arrow if every statement above the line is true. The part above the line is doing most of the work. This describes which conditions have to be true for the rule to be applied, but it also constructs the new feature model with the newly created feature. The reader may notice that we use auxiliary functions such as *isUniqueName* and *addFeatureToGroup*. This is meant for readability, and to avoid cluttering

Operation	**Description**
addFeature(newFid, newName, targetGroup, fType)	Adds the feature with id newFid, name newName, feature type fType to the group targetGroup
removeFeature(featureID)	Removes the feature with ID featureID from the feature table
moveFeature(featureID, newGroupID)	Moves the feature with ID featureID to the group with ID newGroupID
renameFeature(featureID, newName)	Changes name field in the feature with ID featureID to newName
changeFeatureType(featureID, newFeatureType)	Changes the feature variation type of the feature with ID featureID to newFeatureType
addGroup(parentFeatureID, groupID, groupType)	Adds group (groupID, groupType, $\emptyset$) to the list of groups in the feature with ID parentFeatureID.
removeGroup(groupID)	Removes the group with ID groupID from the feature table
changeGroupType(groupID, groupType)	Changes the group variation type of the group with ID groupID to groupType
moveGroup(groupID, newParentFid)	Moves the group with ID groupID to the list of groups in the feature with ID newParentFid

Table 2: Operation overview

the rules with details. The full definition of the SOS rules (and the auxiliary functions) can be found in the appendix.

3.2 Introduction of time

For an analysis of the evolution plans, it is necessary to introduce a concept of time into the semantics. Conceptually, several operations can be performed "at the same time". To tackle this problem, we introduce *time points* and *plans* to associate a group of operations with a time point.

We define a *plan* as a tuple (time point, $\overline{\text{operations}}$), where time point is a natural number, and $\overline{\text{operations}}$ is an ordered list of operations as explained above. We define a *state* as a tuple (time, configuration, $\overline{\text{plans}}$), where $\overline{\text{plans}}$ is a list of plans ordered by time point. No two plans have the same time point; all operations associated with the same time are placed in the same plan. The constructor for plan is ;, which is associative (but *not* commutative). The identity of ; is ρ. A configuration is a feature model followed by an ordered list of operations. The rules will take a configuration FM(RootID, FT) $\langle$Operation$\rangle$ $\overline{\text{Operations}}$ to a configuration FM(RootID, FT') $\overline{\text{Operations}}$. The constructor for a sequence of operations is " " (a space), and the identity of " " is λ. In a valid state, the configuration is on the form FM(RootID, FT) λ, that is, with an empty list of operations. If the initial state is associated with time k, for all $i \geq k$ there are states associated with time i. Following are the SOS rules for advancing time while implementing plans.

$$\frac{\begin{array}{c}\mathrm{FT}(\mathrm{NewFid}) = \bot \\ \mathrm{isUniqueName}(\mathrm{NewName}, \mathrm{FT}) \\ \mathrm{FT}' = \mathrm{addFeatureToGroup}(\mathrm{FT}, \mathrm{TargetGroup}, \mathrm{NewFid}) \\ \mathrm{FT}'' = \mathrm{FT}' + [\mathrm{NewFid} \mapsto (\mathrm{NewName}, \mathrm{TargetFid}, \emptyset, \mathrm{Ftype})] \\ \mathrm{isWellFormed}(\mathrm{FT}'', \mathrm{NewFid})\end{array}}{\begin{array}{c}\mathrm{FM}(\mathrm{RootId}, \mathrm{FT}) \\ \mathrm{addFeature}(\mathrm{NewFid}, \mathrm{NewName}, \mathrm{TargetGroup}, \mathrm{Ftype}) \\ \Longrightarrow \\ \mathrm{FM}(\mathrm{RootId}, \mathrm{FT}'')\end{array}}$$

Figure 5: SOS rule for add feature

Advance time, no plan for current time

$$\frac{\mathrm{CurrentTime} + 1 < \mathrm{NextPlanTime}}{\begin{array}{c}(\mathrm{CurrentTime}, \mathrm{FM}(\mathrm{RootID}, \mathrm{FT})\ \lambda, (\mathrm{NextPlanTime}, \mathrm{Ops}); \mathrm{Plans}) \\ \Longrightarrow \\ (\mathrm{CurrentTime} + 1, \mathrm{FM}(\mathrm{RootID}, \mathrm{FT})\ \lambda, (\mathrm{NextPlanTime}, \mathrm{Ops}); \mathrm{Plans})\end{array}}$$

Advance time, implement plans

$$\frac{\mathrm{CurrentTime} + 1 = \mathrm{NextPlanTime}}{\begin{array}{c}(\mathrm{CurrentTime}, \mathrm{FM}(\mathrm{RootID}, \mathrm{FT})\ \lambda, (\mathrm{NextPlanTime}, \mathrm{Ops}); \mathrm{Plans}) \\ \Longrightarrow \\ (\mathrm{NextPlanTime}, \mathrm{FM}(\mathrm{RootID}, \mathrm{FT})\ \mathrm{Ops}, \mathrm{Plans})\end{array}}$$

Advance time, no more plans

$$\frac{}{\begin{array}{c}(\mathrm{CurrentTime}, \mathrm{FM}(\mathrm{RootID}, \mathrm{FT})\ \lambda, \rho) \\ \Longrightarrow \\ (\mathrm{CurrentTime} + 1, \mathrm{FM}(\mathrm{RootID}, \mathrm{FT})\ \lambda, \rho)\end{array}}$$

Example state

Using the example as shown in Figure 2, the initial state can be expressed in the above defined syntax:

$$
\begin{aligned}
&(0, \mathrm{FM}(\mathrm{C}, \\
&\qquad [\mathrm{C} \mapsto (\mathrm{Car}, \bot, (\mathrm{C1}, \mathrm{AND}, \mathrm{IS}), \mathrm{mandatory})] + \\
&\qquad [\mathrm{IS} \mapsto (\mathrm{Infotainment\ System}, \mathrm{C}, (\mathrm{IS1}, \mathrm{AND}, \mathrm{B}), \mathrm{mandatory})] + \\
&\qquad [\mathrm{B} \mapsto (\mathrm{Bluetooth}, \mathrm{IS}, \emptyset, \mathrm{optional})])\ \lambda,
\end{aligned}
$$

(1, addGroup(IS, IS2, XOR)
addFeature(AA, Android Auto, IS2, optional)
addFeature(ACP, Apple Car Play, IS2, optional));
(2, addFeature(CS, Comfort Systems, C1, optional)
addGroup(CS, CS1, AND)
addFeature(PP, Parking Pilot, CS1, optional)))

After time has advanced to 2, the plans field will contain ρ, and the feature model will be the same as the one in Figure 4, visualized in Figure 1.

3.3 Temporal logic for feature models

We use LTL to capture temporal properties about the feature model. But instead of letting the user specify temporal formulas directly, we instead construct a less expressive intermediate language, that is closer to natural language and what the user want to express.

First we construct the atomic propositions. We make a distinction between *entities*, *events* and *time propositions*, where all of them are atomic propositions.

Entities

Examples of entities are *feature A is mandatory*, or *feature B exists.* To check if *feature B exists*, you simply check if the mapping with id B exists in the feature model in the current state. The syntax for entities are defined as the following (variables are written in **bold**):

- feature **Fid** exists
- group **Gid** exists
- feature **Fid** has type **FType**
- group **Gid** has type **GType**
- feature **Fid** has parent group **Gid**
- feature **Fid** has parent **ParentFid**
- group **Gid** has parent **Fid**
- feature **Fid** has name **Name**

Entities can also be combined using the following four logical connectives.

- not **entity**
- **entity** and **entity**
- **entity** or **entity**
- **entity** implies **entity**

Events

Events are a bit different. Instead of checking if something is true or false in the model, you want to find the exact point of where some entity started to be true, or where it stopped being true. Events are just entities with starts or stops at the end. Examples of this are *feature A exists starts* or *group B is XOR stops*. The syntax for entities are defined as the following:

- **entity** starts

  Example: "feature 3 exists starts". This means that feature 3 was added to the feature model.

- **entity** stops

  Example: "feature 3 exists stops". This means that feature 3 was removed from the feature model.

Similar to entities, events can also be combined using the following four logical connectives.

- not **event**
- **event** and **event**
- **event** or **event**
- **event** implies **event**

Time propositions

Since each state is associated with a time point, we can define some atomic propositions concerning time, called time propositions. Some examples of this are *isTime(3)*, *after(10)* and *before(4)*. These are straightforward atomic properties, because you just have to compare the specified number with the time point in the feature model. The syntax for the atomic properties concerning time are defined as the following:

- isTime(**time point**)
- before(**time point**)
- after(**time point**)

Formulas

With these three types of atomic propositions, you can combine these to make actual valid formulas. These formulas define what queries are valid. Some examples of formulas are “feature A exists at 4”, or “group B exists starts between 3 and feature C exists stops”. The syntax for formulas are defined as the following:

- **entity** at **time point**

  Example: “feature A exists at 3”. Intuitively, this means that at time 3, the feature with ID A has a mapping in the feature table.

- **event** at **time point**

  Example: “feature A has type mandatory starts at 3”. Intuitively, this means that just before 3, the feature with ID A is *not* mandatory, but at time 3, the feature *becomes* mandatory.

- **entity** when **event**

  This has the same meaning as “at”, but instead of explicitly providing a time point, we use an event to give a point in time.

  Example: “feature A exists when feature B has type mandatory starts”. Intuitively, this means at the time point when the feature with ID B becomes mandatory, the feature with ID A has a mapping in the feature table.

- **event** when **event**

  This statement means that two events happen at the same time point.

  Example: “feature A exists starts when feature B has type mandatory starts”. This means that at the time point when the feature with ID B becomes mandatory, the feature with ID A gets a mapping in the feature table (which was not there before).

- **entity** until **time point**

  Example: “feature A exists until 3”. Intuitively, this means that from the initial state until (but not including) the state with time point 3, each state will have a mapping for the feature with ID A. From 3 and on, there may or may not be such a mapping in the feature table.

- **entity** until **event**

  Example: “feature A exists until feature A exists stops”. This essentially has the same meaning as the previous statement, but instead of an explicit time point, we give an event marking the state from which we do not care whether the feature with ID A exists.

- **event** before **time point**

  Example: “feature A exists starts before 2”. Intuitively, this means that feature A gets a mapping in the feature table at a time point that is *strictly less* than 2, for instance 1 or 0.

- **event** before **event**

Example: “feature A exists starts before feature B has type optional starts”. Intuitively, this means that the time point in which feature A starts starts to exist is *strictly less* than the time point in which feature B becomes optional. If these events occur simultaneously, the statement is false.

- **event** after **time point**

  This is the same as *before time point*, except that the event must occur at a time point that is strictly greater than the time point given.

- **event** after **event**

  Example: “feature A exists starts after feature B has type optional starts”. This is equivalent to the statement “feature B has type optional starts before feature A exists starts”.

- **entity** after **time point**

  Example: “feature A exists after 3”. This means that in all states with time points strictly greater than 3, there is a mapping for the feature with ID A in the feature table.

- **entity** after **event**

  Example: “feature A exists after feature A exists starts”. This is similar to the previous statement, but we give an event instead of a time point. Intuitively, it means that in all states after (but not including) the first state where there is a mapping for the feature with ID A in the feature table, the feature table must contain such a mapping.

- **event** between (**time point** | **event**) and (**time point** | **event**)

  Example: “feature A is mandatory starts between 2 and feature B exists stops”. Intuitively, this means that at some point after 2 (including 2), and strictly before feature B is removed from the feature table, feature A becomes mandatory.

- always **entity**

  This is quite straightforward, and expresses invariance.

  Example: “always feature A exists” means that in all states, there is a mapping for the feature with ID A in the feature table.

- **entity** while **entity**

  Example: “not feature B exists while feature A exists”. In all states where there is a mapping for feature A in the feature table, there isn’t one for feature B. If feature A is *not* in the feature table, feature B may or may not be.

- **event** while **entity**

  Example: “feature B starts to exist while feature A exists”. Intuitively, this means that in some state where the feature with ID A has a mapping in the feature table, the feature table gets a mapping for the feature with ID B.

- **entity** while between (**time point** | **event**) and (**time point** | **event**)

Example: "feature A exists while between 2 and 5". In all states from (and including) 2 until (but not including) 5, there is a mapping for feature A in the feature table. These time points may be replaced with events.

4 Implementation in Maude

4.1 Semantics implemented in Maude

Since rewriting logic corresponds closely to structural operational semantics, the logic in the implementation is quite similar to the rules in the formalization. The terms differ only where necessary; some Greek letters have been replaced by keywords in the implementation, and we use Maude's mixfix notation for some of the terms for the sake of readability:

```
sorts FeatureModel FeatureMapping FeatureTable Name .
sorts Feature      FeatureID      FeatureType  Features .
sorts Group        GroupID        GroupType    Groups .

subsort FeatureID      < Features .
subsort Group          < Groups .
subsort FeatureMapping < FeatureTable .

*** Constructors ***
op FM     : FeatureID    FeatureTable -> FeatureModel   [ctor] .
op [_->_] : FeatureID    Feature      -> FeatureMapping [ctor] .
op emptyT :                           -> FeatureTable   [ctor] .
op _+_    : FeatureTable FeatureTable -> FeatureTable   [ctor assoc comm id: emptyT] .

op  f                  : Name FeatureID Groups FeatureType -> Feature [ctor] .
op  noF                :                    -> Features    [ctor] .
op  _::_               : Features Features  -> Features    [ctor assoc comm id: noF] .
ops optional mandatory :                    -> FeatureType [ctor] .

op  g          : GroupID GroupType Features -> Group     [ctor format (r o)] .
op  noG        :                            -> Groups    [ctor format (r o)] .
op  _::_       : Groups  Groups             -> Groups    [ctor assoc comm id: noG] .
ops AND OR XOR :                            -> GroupType [ctor] .
```

Above are the most important of the constructors for the terms in the Maude implementation. The identity element for the feature table is `emptyT` in the implementation, as opposed to ε in the semantics. Instead of using just tuples to model feature and group terms, a feature term (Name, ParentFid, Groups, FType) in the semantics is translated to `f(Name, ParentFid, Groups, FType)` in the Maude implementation. Similarly, a group term (GroupID, GType, Features) is modeled as `g(GroupID, GType, Features)`. Instead of using $\emptyset$ as identity for the `Groups` and `Features` sets, the identities are `noG` and `noF` respectively. A feature mapping [Fid $\mapsto$ (...)] becomes `[Fid -> f(...)]` in Maude.

For simplicity and readability, we use Maude's mixfix notation to construct configurations:

```
op _#_#_ : FeatureModel Operations Log -> Configuration [ctor] .
op done  :                             -> Operations    [ctor] .
op __    : Operations   Operations     -> Operations    [ctor assoc id: done] .
```

The underscores (_) are used to indicate where the arguments should be in the term, so a configuration

$$\text{FM}(\dots)\ \text{addFeature}(\dots)\ \text{removeFeature}(\dots)$$

will look like the `Configuration`

```
FM(...) # addFeature(...) removeFeature(...) # empty
```

The last parameter to the `Configuration` – `Log`, is there for purposes related to temporal logic, and should be ignored for now.

The *states* in the semantics (explained in section 3.2 Introduction of time) are implemented as a sort `TimeConfiguration`. Syntactically, it is different, because we again utilize Maude's mixfix notation for easier readability:

```
op currentTime:_C:_plan:_ : TimePoint Configuration Plans -> TimeConfiguration [ctor] .
op at_do_;                : TimePoint Operations -> Plan         [ctor] .
op end                    :                      -> Plans        [ctor] .
op _;;_                   : Plans Plans          -> Plans        [ctor assoc id: end] .
```

A state

$$(0, \text{FM}(\dots)\ \text{operations}, (1, \text{addFeature}(\dots)\ \text{renameFeature}(\dots)); (5, \text{removeFeature}(\dots)))$$

is implemented in Maude as the `TimeConfiguration`

```
currentTime: 0
C: FM(...) Ops
plan: at 1 do addFeature(...) renameFeature(...) ; ;; at 5 do removeFeature(...) ;
```

The rules for the operations are implemented as equations, where the greatest difference between implementation and formalization is that the pattern matching has been moved to auxiliary functions. This was done for error handling purposes, which are explained in further detail in section 4.2. See the example below of how the rule for `addFeature` and the auxiliary equation `addFeatureToGroup` were implemented as equations in Maude. (See the Github repository[4] for the rest of the auxiliary functions and the other operations).

```
ceq [add-feature] :
  FM(RootFid, FT)
  # addFeature(NewFid, NewName, TargetGroup, FType) Ops
  # L
=
  FM(RootFid, FT'')
```

[4]https://github.com/idamotz/LTEP-summer-project

```
  # Ops
  #
  feature NewFid exists starts ,
  feature NewFid has name NewName starts ,
  feature NewFid has type FType starts ,
  feature NewFid has parent ParentFid starts ,
  feature NewFid has parent group TargetGroup starts ,
  L
if notExists(NewFid, FT)
/\ isUniqueName(NewName, FT)
/\ FT' := addFeatureToGroup(FT, TargetGroup, NewFid)
/\ FT' =/= addFeatureToGroupError
/\ ParentFid := parentOfGroup(FT, TargetGroup)
/\ ParentFid =/= parentOfGroupError
/\ FT'' := FT' + [NewFid -> f(NewName, ParentFid, noG, FType)]
/\ isWellFormed(FT'', NewFid) .

op addFeatureToGroupError : -> FeatureTable .
op addFeatureToGroup :
  FeatureTable GroupID FeatureID -> FeatureTable .
eq addFeatureToGroup(
  FT + [ParentFid -> f(Name', Fid',
    g(Gid, GType, Fs) :: Gs, FType)], Gid, Fid)
= FT + [ParentFid -> f(Name', Fid',
    g(Gid, GType, Fid :: Fs) :: Gs, FType)] .
eq addFeatureToGroup(FT, Gid, Fid)
 = addFeatureToGroupError
    [owise print "ERROR: No feature with group " Gid " exists"] .
```

Figure 6: Code example: addFeature

4.2 Error handling

A large part of the goal is to make sure a provided evolution plan does not contain any paradoxes; for example trying to change the type of a feature after removing it. This is addressed in our implementation using error handling. Maude does not have much support for error handling beyond *stuck terms*, that is, terms that cannot be reduced any further. If a term does not match any pattern in an equation, or if the conditions are not met in a conditional equation, the equation simply is not applied, resulting in unexplained stuck terms. For good error handling, there should be some information about *why* the error occurred, rather than just giving a result containing stuck terms.

To compensate for Maude's poor error handling, we have made use of a combination of `print` and `owise` statements. The `owise` statement lets an equation apply only if no other equations can be applied. Thus, if the pattern matching or other conditions fail in the equation intended for some operation, the `owise` equations can catch this and print the probable cause for failure. To provide good error messages, we decided to move most of the pattern matching used in the formalization to specialized auxiliary equations using the `owise` statement in the Maude implementation. For instance, see the example in Figure 6. The `add-feature` equation uses the auxiliary equation `addFeatureToGroup`, which relies heavily on pattern matching. If the matching fails, it will instead use the `owise` equation, which reduces to the constant `addFeatureToGroupError` and prints an error. The `addFeatureToGroup` equation stops executing on this result, which gives us a more informative stuck state.

Example (addFeature with error)

Given an initial state

```
eq init =
  currentTime: 0
  C: FM("C",
           ["C" -> f("Car", noParent,  g("C1", AND, "IS"), mandatory)] +
           ["IS" -> f("Infotainment System", "C", g("IS1", AND, "B"), mandatory)] +
           ["B" -> f("Bluetooth", "IS", noG, optional)])
     # done
     # empty
  plan: at 1 do
        addFeature("AA", "Android Auto", "IS2", optional)
        addFeature("ACP", "Apple Car Play", "IS2", optional) ; ;;
        at 2 do
        addFeature("CS", "Comfort Systems", "C1", optional)
        addGroup("CS", "CS1", AND)
        addFeature("PP", "Parking Pilot", "CS1", optional) ; .
```

the Maude command `rew init .` will yield the following result:

```
ERROR: No feature with group "IS2" exists
result TimeConfiguration:
currentTime: 1
C:
FM("C", ["B" -> f("Bluetooth", "IS", noG, optional)]
  + ["C" -> f("Car", noParent, g("C1", AND, "IS"), mandatory)]
  + ["IS" -> f("Infotainment System", "C", g("IS1", AND, "B"), mandatory)])
#
addFeature("AA", "Android Auto", "IS2", optional)
addFeature("ACP", "Apple Car Play", "IS2", optional)
#
empty
plan:
at 2 do
addFeature("CS", "Comfort Systems", "C1", optional)
```

```
addGroup("CS", "CS1", AND)
addFeature("PP", "Parking Pilot", "CS1", optional) ;
```

Reading only the result, it is apparent that the program got stuck on the term `addFeature("AA", "Android Auto", "IS2", optional)`. The currentTime field shows us where in the plan the error is, and the error message provides the reason why this did not work; to fix this, we have to add the group `"IS2"` to the feature table before adding Android Auto.

4.3 Equations vs rules - what is a point in time?

When implementing the semantics in Maude, we had to think about granularity of time. If we implemented the rules from the semantics as Maude rules, the temporal logic would be less straightforward. For example, if you were to add feature A and B at time 3, the addition of these features would not happen in the same state, but rather in two separate states. To handle this in a better way, we want states to have a one-to-one correspondence to time points in the plan. To achieve this, each rule is instead implemented as a conditional equation and not a conditional rule. There are only two Maude rules in the implementation, where the main rule, `advance-time` triggers when all the operations from the previous time point has been integrated in the feature model. The rule then moves the next list of operations to the configuration, so that the equations can execute all the operations in the same time point. By carefully choosing what should be equations and what should be rules, we ensure that we get the right granularity of time.

Notice that instead of incrementing time by 1 for each state as in the semantics, our implementation only has states for the time points for which there are plans. The main reason behind this is efficiency. If a user decides to have one plan at time 1 and the next at 1 000 000, it seems unnecessary to have almost a million states in which nothing changes except the current time. Instead, we keep only the two states, but the interpretation is the same.

```
rl [advance-time] :
    currentTime: CT C: (FM(RootFid, FT) # done # L)
    plan: at TP do Ops ; ;; Ps
  =>
    currentTime: TP C: (FM(RootFid, FT) # Ops  # empty)
    plan:                     Ps .
```

Where the rules in the formalization lets the time increment for all eternity, the Maude implementation instead has an additional rule `finish-plan` which changes the the `currentTime` field to a constant `endTime`, signifying that there will be no more change to the feature model after this point.

```
rl [finish-plan] :
    currentTime: CT      C: FM(RootFid, FT) # done # L     plan: end
  =>
    currentTime: endTime C: FM(RootFid, FT) # done # empty plan: end .
```

Using this, we can simulate an endless sequence of states for each point in time without actually having to store all these states. This makes model checking much simpler, because we have a finite number of states to analyze.

4.4 Model checking in Maude

Assumptions

We assume that before using the model checker on a plan, the plan has been executed once in the Maude specification, and does not contain any paradoxes. Furthermore, we assume that no events can happen twice; i.e. a feature cannot be added, then removed, and then added again. We also assume that all the plans for a single time point are collected into one `Plan` term, so that each time point only matches a single state.

Entities

The implementation of the entities was pretty straightforward, since all we really had to do was to check if some property was true in the feature model. Since we also support composing entities with logical operators, we defined a helper function ⊢ to handle the cases with entities combined with logical operators. For example, the equations relevant to model check the entity "feature A exists and feature B exists" are shown below. (note that the operation on entities is called `andalso`, since Maude's built-in boolean operator is called `and`).

```
--- To model check any entity, call the helper function |-
eq currentTime: Time C: C plan: Ps |= Ent  = C |- Ent .
--- andalso-entities is deconstructed into checking both left and right hand side
eq C |- Ent andalso Ent' = C |- Ent and C |- Ent' .
--- if the mapping with id Fid exist, proposition is true
eq FM(RootFid, [Fid -> F] + FT) # Ops # L
   |- feature Fid exists = true .
--- otherwise, the feature does not exist, and the proposition is false
eq C |- Ent = false [owise] .
```

Events

Checking the entities is pretty straightforward, because you can just look at the feature model to determine their truth value, but checking when an entity starts to be true or stops being true is more complicated. You can't just look at the feature model in a single state to determine this, because you have to know that there was an actual change from the previous state. Since the model checker in Maude can't look at the previous state, implementing the events directly would result in very messy formulas. To make events into state propositions, we instead track every change in the feature model, by adding a log of events to each configuration. Each state then has a log of every change made in that state, which makes model checking events pretty simple. As seen in the rule for add feature, Figure 6, the events are stated explicitly. The configuration after executing the add-feature equation has several events attached to it, including "feature NewFid exists starts", "feature NewFid has name starts“, etc. The only two equations needed for model checking events are the following:

```
--- Simply checks that an events exists in the log for a configuration
eq Model # Ops # Ev , L
   |- Ev = true .
```

```
--- The owise equation applies only if no other equations can apply
eq C |- Ev = false [owise] .
```

Time Propositions

The implementation of time propositions differs a bit from the semantics, because of the possibility of having gaps in the timeline. For example, if you have plan that adds a feature A at time 2, and a feature B added at time 4, there will not exist a state with time 3. To handle this, we chose to have two different implementations of `isTime`; `isTime` (explicit) and `isTimeImplicit`. In this example, `isTime(3)` will not be true for any of the three states, because there are only states for time points 0, 2 and 4. In the case of `isTimeImplicit`, `isTimeImplicit(3)` will return true in the state with time 2 only. In the implementation of the `after`-proposition, we explicitly state that if we are in the special state with the time point `endTime`, we are in fact always after any time point. Some of the time propositions are shown below.

```
--- the time points has to be exactly equal
eq currentTime: TP C: C plan: Ps
   |= isTime(TP) = true .

--- one case for normal states, one case for the state with endTime
eq currentTime: CT C: C plan: Ps
   |= after(TP) = CT > TP .
eq currentTime: endTime C: C plan: Ps
   |= after(TP) = true .

--- Either TP is true in one of the states, or this is true for the
--- state directly before TP.
eq isTimeImplicit(TP) = isTime(TP) \/ (before(TP) /\ O after(TP)) .
```

Events vs Entities

Having gaps in the timeline also has implications for the implementation details regarding formulas with entities and events, because the two are fundamentally different. Entities are true over continuous intervals, but events are only true at a point in time. This difference is clear when you have a plan where there are gaps between the time points. For example if you add feature A at 2, and add feature B at 4, what will happen if you make queries that reference time 3? If you ask if feature A exists at time 3, it should be true. If you ask if feature A exists started at time 3, you should get false, because it actually started at 2. However, there are only 3 time points in the evolution, 0 (initial state), 2 and 4, so there is no state with time 3. This implies that entities and events should be treated differently in the rules, based on what makes sense. The two formulas in the example below demonstrates this difference.

```
eq Ent at TP = <> (isTimeImplicit(TP) /\ Ent) .
eq Ev  at TP = <> (isTime(TP)         /\ Ev)  .
```

Formulas

Implementing the formulas was just a matter of using the temporal logic formulas together with the *entities*, *events* and *time propositions*. The formulas are written as equations, where you have one equation for each valid combination of atomic propositions. As an example, the formulas for *after* are implemented as the following.

```
--- There exists a state where Ev is true and after(TP) is true
eq Ev  after TP  = <> (Ev  /\ after(TP)) .
--- flips the arguments and uses the formula for before
eq Ev  after Ev' =     Ev' before Ev     .
--- it is always such that if you are after TP, Ent is true.
eq Ent after TP  = [] (after(TP) -> Ent) .
--- At some point, Ev happens and Ent is always true from the next state and beyond
eq Ent after Ev  = <> (Ev /\ O [] Ent)   .
```

4.5 Usage

If you are using a feature model to develop a software product line, you can use our system to verify that your plan is correct, and to see what your feature model will look like at a future time. To use the system, construct a Maude module on this form:

```
load featuremodel-checker.maude

mod MY-PLAN is including FEATUREMODEL-CHECKER . protecting STRING + NAT .
    --- You can use String as the sort for FeatureID, GroupID, and Name.
    --- Could also have been Qid or Int (or anything else).
    subsorts String < FeatureID GroupID Name .
    --- Create a constant for the parent of the root.
    op noParent : -> FeatureID .
    --- Create initial configuration (what your feature model looks like now)
    op configuration : -> Configuration .
    eq configuration = FM("RootID",
                          ["RootID" -> f("RootName", noParent, noG, mandatory)])
                          # done
                          # empty . --- Example configuration.
    --- Model plan (which changes you plan on making, and when)
    op plan : -> Plans .
    eq plan = at 1 do --- specify time for the plan, must be natural number
              --- add a group to the root feature
              addGroup("RootID", "Group1", AND)
              --- add a feature to the newly created group
              addFeature("ChildID", "ChildName", "Group1", optional) ; ;;
              at 5 do
              --- Give the new feature a better name
```

```
            renameFeature("ChildID", "BetterName") ; .

    --- Use plan and configuration to make an initial state for the model checker
    op init : -> TimeConfiguration .
    eq init = currentTime: 0 C: configuration plan: plan .
endm
```

Next, run Maude and input `in <filename.maude>`. Next, check that your plan runs without errors:

```
Maude> rew init .
```

The result should be

```
rewrite in MY-PLAN : init .
result TimeConfiguration:
currentTime: endTime
C:
FM("RootID", ["ChildID" -> f("BetterName", "RootID", noG, optional)]
  + ["RootID" -> f("RootName", noParent, g("Group1", AND, "ChildID"),
    mandatory)])
#
done
#
empty
plan: end
```

Here, we can see the final state of the feature model, with all the plans implemented. If it gets stuck, it will give an error message showing why, which can be used to correct the plan. If the plan is correct, like it is here, you can test temporal constraints:

```
Maude> red modelCheck(init, feature "ChildID" exists after 1) .
reduce in MY-PLAN : modelCheck(init, feature "ChildID" exists after 1) .
result Bool: true
```

If the constraint fails, it will give a counterexample, showing the path where the formula failed.

4.6 Limitations of the model checker

Although the model checker we have implemented will output whether a temporal constraint holds, it does not give much information about why a constraint does not hold. A counterexample consists of all the states of the execution, and it can be quite difficult to locate exactly where the condition failed. This is something that should be improved if we continue to develop this tool, as a user will want to correct errors as well as detecting them.

The model checker also does not accept queries like ”When does feature 3 exist?” or ”What are the child features of feature 2 at time 6?”, because it can only check if something is true or false for the plan. If the tool were to be extended, these are a few of the aspects that we would want to improve upon.

4.7 Determinism and termination

It would seem that our specification is both deterministic and terminating, given a valid initial state (i.e. no loops in the feature table). Given a state and a list of operations, there is only one possible result; after the operations are executed, either the plan has been implemented and we have a final result, or the plan contains an error and we reach a stuck state. In both cases, the specification terminates. Because of this, we have no non-determinism or infinite branches in our program, which means that we do not yet make full use of Maude's model checker, which can handle both. Of course, there is no law that says that to utilize the model checker you need to have an immensely complex system, but maybe some other tool could have been used to implement the model at least as efficiently. However, in future there may be introduced non-determinism, for example in generating new plans that fix errors, and in that case Maude is a very good tool.

5 Conclusion

Software product lines and highly configurable systems are widely used, and there is a need for good evolution planning over time. Using the Maude system, we have developed a tool which may help in making structured and well-formed plans for the long-term evolution of feature models and software product lines. The tool can be used to verify that a plan is correct and has no errors, as well as to assure that long-term goals for a project are met.

In the future, this tool may be integrated into existing systems for development in planning and implementing feature models. The Maude specification could also be extended to include non-determinism and more sophisticated error handling, for example giving possible solutions to fix errors in a plan.

References

[1] José Meseguer. Twenty years of rewriting logic. *J. Log. Algebr. Program.*, 81(7-8):721–781, 2012.

[2] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications - a formal introduction.* Wiley professional computing. Wiley, 1992.

[3] Peter Csaba Ölveczky. *Designing Reliable Distributed Systems - A Formal Methods Approach Based on Executable Modeling in Maude.* Undergraduate Topics in Computer Science. Springer, 2017.

[4] Christoph Seidl. *Integrated Management of Variability in Space and Time in Software Families.* PhD thesis, Dresden University of Technology, Germany, 2017.

Appendix A Semantics of feature model evolution plans

We define the terms of a feature model, a feature table, a feature group, and constructors and functions as the following:

A feature model is captured by a term, i.e. $FM(RootId, FT)$, which contains the root ID of the feature model and the corresponding feature table (FT).

A feature table is a mapping, which maps each feature ID (Fid) to a term containing the feature name, the parent feature of the feature, a set of feature groups that belong to the feature, and the type of the feature, i.e. $[Fid \mapsto (name, parent, \overline{group}, Ftype)]$. The feature types are optional and mandatory. A feature table is always non-empty and at least contains the mapping of the root. If a feature table FT contains mapping $[Fid \mapsto (name, parent, \overline{group}, Ftype)]$, then $FT(Fid) = (name, parent, \overline{group}, Ftype)$.

A feature group is defined as a term containing the group ID, the type of group, and a set of feature IDs, i.e. $group := (id, Gtype, \overline{featureId})$. The group types are AND, OR, and XOR, where an AND group can contain both optional and mandatory features; an OR group contains only optional features where several features can be selected at the same time; and an XOR group contains optional features, where exactly one feature must be selected.

We define $+$ as constructor for feature table and :: as constructor for set, where ε is the identity element for $+$, and $\emptyset$ is the identity element for ::. Both are commutative and associative.

Well-formedness

We define a function for checking if the combination of feature type and its parent group type is well formed.

```
isWellFormed(FT, Fid) =
      ¬(Type = mandatory ∧ Gtype ≠ AND)
    ∧ ¬(Type ≠ optional ∧ (Gtype = OR ∨ Gtype = XOR))
    where FT = FT'
             + [Fid ↦ (_, ParentFid, _, Type)]
             + [ParentFid ↦ (_, _, (_, Gtype, Fid::_)::_, _)]
```

Add Feature

```
addFeatureToGroup(FT, TargetGroup, NewFid) =
    FT' + [TargetFid ↦ (TargetName,
        ParentOfTarget,
        (TargetGroup, Gtype, NewFid::Features)::Groups,
        Ftype)]
    where FT = FT' + [TargetFid ↦ (TargetName,
        ParentOfTarget,
        (TargetGroup, Gtype, Features)::Groups,
        Ftype)]

isUniqueName(name, ε) = true
isUniqueName(name, FT + [id ↦ (name', _, _, _)]) =
    name ≠ name' ∧ isUniqueName(name, FT)
```

$$\frac{\begin{array}{c}\text{FT(NewFid)} = \bot \\ \text{isUniqueName(NewName, FT)} \\ \text{FT}' = \text{addFeatureToGroup(FT, TargetGroup, NewFid)} \\ \text{FT}'' = \text{FT}' + [\text{NewFid} \mapsto (\text{NewName, TargetFid}, \emptyset, \text{Ftype})] \\ \text{isWellFormed}(\text{FT}'', \text{NewFid})\end{array}}{\begin{array}{c}\text{FM(RootId, FT)} \\ \text{addFeature(NewFid, NewName, TargetGroup, Ftype)} \\ \Longrightarrow \\ \text{FM}(\text{RootId}, \text{FT}'')\end{array}}$$

Remove Feature

```
removeFeatureFromParent(FT, ParentFid, RemoveFid) =
    FT' + [ParentFid ↦ (TargetName,
        ParentOfTarget,
        (TargetGroup, Gtype, Features)::Groups,
        Ftype)]
    where FT = FT' + [ParentFid ↦ (TargetName,
        ParentOfTarget,
        (TargetGroup, Gtype, RemoveFid::Features)::Groups,
        Ftype)]
```

$$\frac{\begin{array}{c}\text{RemoveFid} \neq \text{RootId} \\ \text{FT} = \text{FT}' + [\text{RemoveFid} \mapsto (\text{Name, ParentFid}, \emptyset, \_)] \\ \text{FT}'' = \text{removeFeatureFromParent}(\text{FT}', \text{ParentFid, RemoveFid})\end{array}}{\begin{array}{c}\text{FM(RootId, FT)} \\ \text{removeFeature(RemoveFid)} \\ \Longrightarrow \\ \text{FM}(\text{RootId}, \text{FT}'')\end{array}}$$

Move Feature

```
isSubFeature(TargetFid, RootId, RootId, FT) = False
isSubFeature(TargetFid, TargetFid, RootId, FT) = True
isSubFeature(TargetFid, TempFid, RootId, FT) =
    isSubFeature(TargetFid, Parent, RootId, FT)
    where FT = FT' + [TempFid ↦ (_, Parent, _, _)]
```

```
parentOfGroup(FT, GroupID) = ParentFid
    where FT = FT' + [ParentFid ↦ (_, _, (GroupID, _, _)::Groups, _)]
```

$$\frac{\begin{array}{c}\text{RootId} \neq \text{MoveFid} \\ \neg\text{isSubFeature(MoveFid, NewParent, RootId, FT)} \\ \text{NewParent} = \text{parentOfGroup(FT, NewGroup)} \\ \text{FT} = \text{FT}' + [\text{MoveFid} \mapsto \text{(Name, ParentFid, Groups, Type)}] \\ \text{FT}'' = \text{removeFeatureFromParent}(\text{FT}', \text{ParentFid, MoveFid}) \\ \text{FT}''' = \text{addFeatureToGroup}(\text{FT}'', \text{NewGroup, MoveFid}) \\ \text{isWellFormed}(\text{FT}''', \text{MoveFid})\end{array}}{\begin{array}{c}\text{FM(RootId, FT)} \\ \text{moveFeature(MoveFid, NewGroup)} \\ \Longrightarrow \\ \text{FM}(\text{RootId, FT}''' + [\text{MoveFid} \mapsto \text{(Name, NewParent, Groups, Type)}])\end{array}}$$

Rename Feature

$$\frac{\begin{array}{c}\text{isUniqueName(NewName, FT)} \\ \text{FT} = \text{FT}' + [\text{TargetFid} \mapsto \text{(Name, Parent, Groups, Type)}] \\ \text{FT}''= \text{FT}' + [\text{TargetFid} \mapsto \text{(NewName, Parent, Groups, Type)}]\end{array}}{\begin{array}{c}\text{FM(RootId, FT)} \\ \text{renameFeature(TargetFid, NewName)} \\ \Longrightarrow \\ \text{FM}(\text{RootId, FT}'')\end{array}}$$

Change Feature Variation Type

$$\frac{\begin{array}{c}\text{RootId} \neq \text{TargetFid} \\ \text{FT} = \text{FT}' + [\text{TargetFid} \mapsto \text{(Name, Parent, Groups, Type)}] \\ \text{FT}''= \text{FT}' + [\text{TargetFid} \mapsto \text{(Name, Parent, Groups, NewType)}] \\ \text{isWellFormed}(\text{FT}'', \text{TargetFid})\end{array}}{\begin{array}{c}\text{FM(RootId, FT)} \\ \text{changeFeatureType(TargetFid, NewType)} \\ \Longrightarrow \\ \text{FM}(\text{RootId, FT}'')\end{array}}$$

Note:

- changing feature type from mandatory to optional is always allowed
- to change feature type from optional to mandatory, the user needs to modify the group type to AND first

Add Group

```
isUniqueGroupId(groupId, ε) =
    true
isUniqueGroupId(groupId, [Fid ↦ (_, _, groups, _)] + FT) =
    ¬contains(groups, groupId) ∧ isUniqueGroupId(groupId, FT)

contains(∅, groupId) = False
contains((id, _, _)::groups, groupId) =
    id == groupId ∨ contains(groups, groupId)
```

$$\frac{\begin{array}{c}\text{isUniqueGroupId(GroupId,FT)}\\ \text{FT} = \text{FT}' + [\text{TargetFid} \mapsto (\text{Name, Parent, Groups, Type})]\\ \text{FT}'' = \text{FT}' + [\text{TargetFid} \mapsto (\text{Name, Parent, (GroupId, GroupType, } \emptyset\text{)::Groups, Type})]\end{array}}{\begin{array}{c}\text{FM(RootId, FT)}\\ \text{addGroup(TargetFid, GroupId, GroupType)}\\ \Longrightarrow\\ \text{FM(RootId, FT}''\text{)}\end{array}}$$

Remove Group

$$\frac{\begin{array}{c}\text{FT} = \text{FT}' + [\text{TargetFid} \mapsto (\text{Name, Parent, (GroupId, GroupType, } \emptyset\text{)::Groups, Type})]\\ \text{FT}'' = \text{FT}' + [\text{TargetFid} \mapsto (\text{Name, Parent, Groups, Type})]\end{array}}{\begin{array}{c}\text{FM(RootId, FT)}\\ \text{removeGroup(GroupId)}\\ \Longrightarrow\\ \text{FM(RootId, FT}''\text{)}\end{array}}$$

Change Group Variation Type

$$\frac{\begin{array}{c}\text{FT} = \text{FT}' + [\text{ParentFid} \mapsto (\text{Name, Parent, (GroupId, Type, Features)::Groups, Type})]\\ \text{FT}''= \text{FT}' + [\text{ParentFid} \mapsto (\text{Name, Parent, (GroupId, NewType, Features)::Groups, Type})]\\ \forall \text{feature} \in \text{Features} \cdot \text{isWellFormed(FT}'', \text{feature)}\end{array}}{\begin{array}{c}\text{FM(RootId, FT)}\\ \text{changeGroupType(GroupId, NewType)}\\ \Longrightarrow\\ \text{FM(RootId, FT}''\text{)}\end{array}}$$

Note:

- changing group type from OR/XOR to AND is always allowed
- to change group type from AND to OR/XOR, the user needs to modify the feature type of all the features belong to the group to optional first

Move Group

```
updateParents(FT, ∅, toFid) = FT
updateParents(FT, fid::features, toFid) =
    updateParents(
        (FT' + [fid ↦ (name, toFid, groups, type)]),
        features,
        toFid)
    where FT = FT' + [fid ↦ (name, _, groups, type)]
```

$$\frac{\begin{array}{c}\forall \text{feature} \in \text{Features} \cdot \neg\text{isSubFeature}(\text{feature}, \text{newParentFid}, \text{RootId}, \text{FT}) \\ \text{FT} = \text{FT}' + [\text{OldParentFid} \mapsto (\text{Name}', \text{Parent}', (\text{GroupID}, \text{GroupType}, \text{Features})::\text{Groups}', \text{Type}')] \\ + [\text{NewParentFid} \mapsto (\text{Name}, \text{Parent}, \text{Groups}, \text{Type})] \\ \text{FT}'' = \text{FT}' + [\text{OldParentFid} \mapsto (\text{Name}', \text{Parent}', \text{Groups}', \text{Type}')] \\ + [\text{NewParentFid} \mapsto (\text{Name}, \text{Parent}, (\text{GroupID}, \text{GroupType}, \text{Features})::\text{Groups}, \text{Type})] \\ \text{FT}''' = \text{updateParents}(\text{FT}'', \text{Features}, \text{NewParentFid})\end{array}}{\begin{array}{c}\text{FM}(\text{RootId}, \text{FT}) \\ \text{moveGroup}(\text{GroupID}, \text{NewParentFid}) \\ \Longrightarrow \\ \text{FM}(\text{RootId}, \text{FT}''')\end{array}}$$

Configuration

We define a configuration as a feature model followed by an ordered list of operations. The previously defined rules will take a configuration FM(RootID, FT) $\langle$Operation$\rangle$ $\overline{\text{Operations}}$ to a configuration FM(RootID, FT') $\overline{\text{Operations}}$. The operator for operations is “ ” (a space), which is associative (but *not* commutative). The identity of “ ” is λ.

State

We define a *plan* as a tuple (time point, $\overline{\text{operations}}$), where time point is a natural number, and $\overline{\text{operations}}$ is an ordered list of operations as explained above. We define a *state* as a tuple (time, configuration, $\overline{\text{plans}}$), where $\overline{\text{plans}}$ is a list of plans ordered by time point. No two plans have the same time point; all operations associated with the same time are placed in the same plan. The constructor for plan is ;, which is associative (but *not* commutative). The identity of ; is ρ. In a valid state, configuration is on the form FM(RootID, FT) λ, that is, with an empty list of operations. If the initial state is associated with time k, for all $i \geq k$ there are states associated with time i.

Advance time, no plan for current time

$$\frac{\text{CurrentTime} + 1 < \text{NextPlanTime}}{\begin{array}{c}(\text{CurrentTime, FM(RootID, FT)}\ \lambda\text{, (NextPlanTime, Ops); Plans)}\\ \Longrightarrow \\ (\text{CurrentTime} + 1\text{, FM(RootID, FT)}\ \lambda\text{,(NextPlanTime, Ops); Plans)}\end{array}}$$

Advance time, implement plans

$$\frac{\text{CurrentTime} + 1 = \text{NextPlanTime}}{\begin{array}{c}(\text{CurrentTime, FM(RootID, FT)}\ \lambda\text{, (NextPlanTime, Ops); Plans)}\\ \Longrightarrow \\ (\text{NextPlanTime, FM(RootID, FT) Ops, Plans})\end{array}}$$

Advance time, no more plans

$$\frac{}{\begin{array}{c}(\text{CurrentTime, FM(RootID, FT)}\ \lambda,\ \rho)\\ \Longrightarrow \\ (\text{CurrentTime} + 1\text{, FM(RootID, FT)}\ \lambda,\ \rho)\end{array}}$$