Master Essay

Ida Sandberg Motzfeldt

August 20, 2020

Contents

Chapter 1

Introduction

Software engineering methodologies for highly-variable software systems lack support for planning the long-term evolution of software. We address this lack in the context of software product lines.

While concepts exist for the construction of software product lines (SPLs), evolution is performed mostly as an informal procedure relying on the intuition and experience of individual engineers with, at most, medium-term goals in mind. Software product line evolution is a major challenge in SPL engineering as many stakeholders are involved, many requirements exist, and changing the SPL potentially influences many configurations. The lack of long-term planning for SPL evolution creates a risk of significantly increased development costs, deviation from intended development direction in collaborative efforts and, ultimately, missing long-term goals, which potentially causes a loss of clients and market shares for SPL vendors due to not addressing market needs properly.

Feature models structure SPLs as trees, giving relations and dependencies between features. They are used widely to model and reason about software product lines, as they model precisely both the variability and unchanging parts of a software product line. When inserting new planned modifications in between the current and a future planned state of a feature model , these changes may cause the original evolution goal to no longer be reachable, either because of domain-related problems or because the changes destroy the structure of the feature model tree. The focus of this thesis are the paradoxes that occur *structurally* based on the semantics of feature model; i.e. when the model is not structurally valid at some point in time as a result of replanning. . Detecting these evolution paradoxes is challenging due to the inherent complexity of the configuration knowledge but is made significantly more complex due to the notion of time present in evolution planning.

To counter these issues, we devise an approach for allowing dynamic changes to an evolution plan while being able to identify evolution paradoxes caused in the future. To achieve this, we will define a formal semantics for how evolution plans can be changed and develop a modular static analysis method to guarantee soundness of the changes.

Chapter 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 software artifact, called a *variant*, is defined by a configuration that consists of a set of selected features.

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.

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 2.1 on the following page for an example of a (simplified) 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

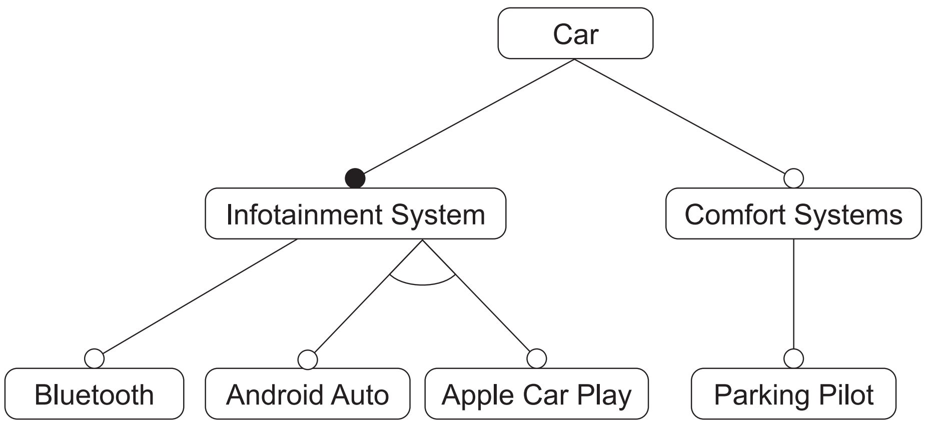


Figure 2.1: Example feature model

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.

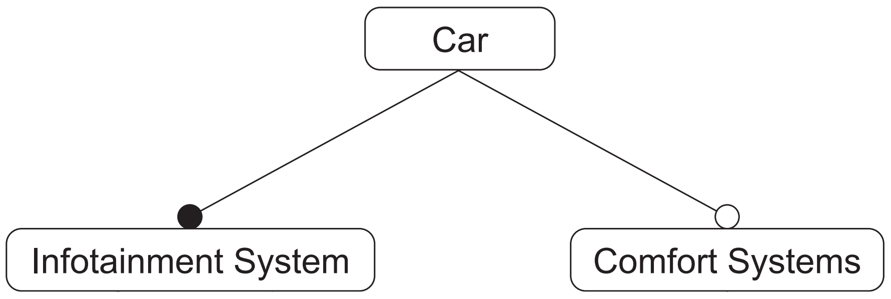
There are some rules for feature models:

- 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.

2.3 Evolution planning

The current tools only deal with the construction of SPLs, and not the planned long-term evolution. On large software product lines, 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 an *AND* group to `Infotainment system`
add feature `Bluetooth` to the `Infotainment System` *AND* group
add an *AND* group to `Comfort Systems`
add feature `Parking Pilot` to the `Comfort Systems` *AND* group

Figure 2.2: Example evolution plan

planning of long term evolution is lacking, and thus creating a risk of increased development costs.

An evolution plan is 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 2.2.

Implementing this plan results in the feature model shown in figure 2.1 on the preceding page.

2.3.1 Changing a plan

As with most large-scale projects, it is often necessary to change the evolution plan for an SPL along the way, due to new requirements or unforeseen circumstances. Evolution plans for SPLs can grow very large and complex, and consequently it can be challenging to discover paradoxes resulting from plan changes. It is therefore crucial that there exist support to integrate automatic paradox detection into the development process , and the solution must be efficient and scalable due to the size and complexity of software product lines.

2.3.2 Example evolution paradox

Due to financial difficulties, an example car manufacturer is obliged to cut costs for their planned car SPL (see figure 2.2 on the previous page). As it results from a short-notice decision, the car system is supposed to drop its stand-alone `Infotainment` system. Instead, the car manufacturer decides to focus on a docking station in combination with more sophisticated `Bluetooth` functionality. For the FM evolution plan, this requires replanning activity: The manufacturer retroactively plans to delete the feature `Infotainment` at the new intermediate time point $t_0.5$, after the initial version for t_0 and before the originally planned version for t_1.

It is important to note that retroactively inserting intermediate edit operations entails specific challenges due to the different orders of devising and scheduling changes. When devising the changes of the example, their order is as follows:

t_0: Plan initial car system.

t_1: Plan to add `Bluetooth` feature as child of `Infotainment`.

$t_0.5$: Replan to delete `Infotainment`.

In contrast, the order in which these changes are scheduled (supposed to be implemented) is the temporal order of all planned changes:

t_0: Plan initial car system.

$t_0.5$: Replan to delete `Infotainment`.

t_1: Plan to add `Bluetooth` feature as child of `Infotainment`.

However, incorporating subsequently scheduled edit operations can fail and, thus, damage the structural consistency of an FM evolution plan: In the car example, the planned change for t_1 to add `Bluetooth` as child feature of `Infotainment` can no longer be implemented as, once reaching t_1, the `Infotainment` feature will have been deleted in the previous $t_0.5$. The reason for this problem is that the replanned FM evolution plan is structurally inconsistent as an evolution paradox was introduced into the evolution plan of the car SPL with the retroactive intermediate change at $t_0.5$.

2.4 Feature model semantics

In this section, we give a formal definition of feature models, the basic evolution operations on feature models and the operational semantics of evolution plans.

Feature model formalization

A feature model is a term **FM(rootID, FT)**, where **rootID** is the root feature's ID, and **FT** is a feature table. A feature table contains mappings from feature IDs to feature terms.

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

where **name** is the name of the feature, **parentFid** is the parent feature ID, $\overline{\textbf{groups}}$ is a set of groups directly below the feature, and **featureType** is the variation type of the feature (i.e., optional or mandatory). A group is defined as a tuple **(groupID, groupType,** $\overline{\textbf{features}}$**)**, where **groupID** is the group ID, **groupType** is the variation type of the group (i.e., OR, XOR or AND), and $\overline{\textbf{features}}$ is a set of feature IDs belonging to the group. We use + as the constructor for the feature table. It is a commutative and associative operation with the identity element ϵ.

Formal representation of feature model

Using the feature IDs *car*, *infotainment*, *bluetooth*, *auto*, *carPlay*, *assistance*, and *pilot*, as well as the group IDs *car1*, *info1*, *info2*, and *comfort1*, we formalize the feature model state of Figure 2.1 on page 4 as:

$$
\begin{aligned}
&\mathrm{FM(car,}\\
&\qquad [\mathrm{car} \mapsto (\mathrm{Car}, \bot, (\mathrm{car1}, \mathrm{AND}, \mathrm{infotainment} :: \mathrm{assistance}), \mathrm{mandatory})] +\\
&\qquad [\mathrm{infotainment} \mapsto (\mathrm{Infotainment\ System}, \mathrm{car}, (\mathrm{info1}, \mathrm{AND}, \mathrm{bluetooth})\\
&\qquad\ :: (\mathrm{info2}, \mathrm{XOR}, \mathrm{auto}\\
&\qquad\ :: \mathrm{carPlay}), \mathrm{mandatory})] +\\
&\qquad [\mathrm{bluetooth} \mapsto (\mathrm{Bluetooth}, \mathrm{infotainment}, \emptyset, \mathrm{optional})] +\\
&\qquad [\mathrm{auto} \mapsto (\mathrm{Android\ Auto}, \mathrm{infotainment}, \emptyset, \mathrm{optional})] +\\
&\qquad [\mathrm{carPlay} \mapsto (\mathrm{Apple\ Car\ Play}, \mathrm{infotainment}, \emptyset, \mathrm{optional})] +\\
&\qquad [\mathrm{assistance} \mapsto (\mathrm{Comfort\ Systems}, \mathrm{car}, (\mathrm{comfort1}, \mathrm{AND}, \mathrm{pilot}),\\
&\qquad \mathrm{optional})] +\\
&\qquad [\mathrm{pilot} \mapsto (\mathrm{Parking\ Pilot}, \mathrm{assistance}, \emptyset, \mathrm{optional})])
\end{aligned}
$$

Table 2.1 on the next page describes the basic operations for planning the evolution of feature models. These operations range from non-disruptive feature model extentions to considerable modifications such as relocating and removing features and groups. An evolution plan consists of an ordered sequence of such operations, i.e., Op_1 Op_2 ... Op_n where each Op_i is an evolution operation.

Operation	Description
addFeature(fid, name, gid, type)	**Add a new feature to a target group** with a given feature id, feature name, group id, and feature type
removeFeature(fid)	**Remove a feature from the feature model** with a given feature id
moveFeature(fid, gid)	**Move a feature to another group** with a given feature id and group id
renameFeature(fid, name)	**Rename a feature** with a given feature id and a new feature name
changeFeatureType(fid, type)	**Change the feature variation type** with a given feature id and a new feature type
addGroup(fid, gid, type)	**Create a new group and add it to a feature** with a given feature id, group id, and group type
removeGroup(gid)	**Remove a group from the feature model** with a given group id
changeGroupType(gid, type)	**Change the group variation type** with a given group id and a new group type
moveGroup(gid, fid)	**Move a group to another feature** with a given group id and feature id

Table 2.1: The basic operations for extending and modifying feature models

2.4.1 Structural operational semantics for feature models

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)
```

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

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)]
```

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

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, _)]
```

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

Rename Feature

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

Change Feature Variation Type

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

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)]
```

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

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

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

Advance time, implement plans

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

Advance time, no more plans

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

2.5 Static analysis

In *Principles of program analysis* (1999), Nielson et al. give the following introduction to static analysis:

> Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer. ([4])

There are various methods and applications for static analysis, including Data Flow Analysis, Control Flow Analysis, Abstract Interpretation, and Type and Effect Systems [4]. Static analysis provides methodologies for validating software. Due to the inherent undecidability of program behaviour, the methods tend to either give false positives (sacrificing soundness), or incomplete but sound answers. Here the focus will be *syntax-driven semantic analysis*, which is expanded on in section 2.5.1 on the following page.

2.5.1 Syntax-driven semantic analysis

The literature does not give a single unified definition of syntax-driven semantic analysis, but there seem to be common elements across the various applications of it. The applications include linguistics (most commonly) and program analysis.

In general, syntax-driven semantic analysis takes into account only the structure (grammar) of the programming language in question, with semantic attachments to this [3]. A program is then parsed, creating a parse tree, which can then be analyzed bottom-up by using the semantic attachments to the grammar[1, 2].

Chapter 3

My project/Challenge

We will use techniques from static analysis to detect paradoxes in feature model evolution plans. The goal is to analyze only the parts of the plan that *may* be affected by change to detect paradoxes, thereby reducing the execution time for verification per plan change. The feature model evolution plan will be treated as a program, and the static analysis techniques will be developed to accommodate this. Feature model evolution plans have some nice properties which programs in general do not have: Since the feature model and plan semantics do not contain branching, the execution of a feature model evolution plan contains only a single branch; this removes a layer of complexity from the task.

3.1 Why existing solutions are not adequate

Although there are tools for planning SPL evolution, they do not have sufficient support for *re*planning. DarwinSPL only detects paradoxes in the last state , and does not allow for changing intermediate states. The Maude solution does detect paradoxes, but it needs to go through the entire plan each time the plan is changed. When a plan grows large, this approach becomes impractical. Another disadvantage to the Maude solution is that it does not find the origin of a paradox (the change that resulted in a paradox), only the paradox itself. A better solution would be a more incremental approach, which builds the analysis on previous results. Given a plan and a plan change, this analysis would assume that the original plan is sound because it has already been checked, and re-check only the parts of the plan affected by the change. For instance, if the change is to remove a feature, it is only necessary to check whether the feature is changed later in the future, e.g. adding a child feature or renaming the feature.

3.2 Dependency bookkeeping

The representation of feature model evolution plans defined in the semantics (see section 2.4 on page 6) is not enough in itself for the purpose of static analysis. Locating all the information about one feature is linear, and given the number of features and groups that may be affected by a plan change, the lookup should be constant. For this it is better to keep a map from the feature IDs to *validities*, that is, intervals when different properties hold for the feature - e.g. feature A has parent feature B holds in $[T_0 - T_1]$, where T_0 and T_1 are time points, A and B are feature IDs, and $[T_0 - T_1]$ is the interval in which feature A has parent feature B. Then the properties which must hold, e.g. that feature A always has a parent, can easily be checked: The validity for feature A's parent must contain the validity for feature A.

This bookkeeping is analogous to the way Bianculli et al.[1, 2] use attributes to incrementally verify programs, capitalizing on previous results of the analysis to efficiently isolate the parts of the program that need to be re-evaluated after any change.

In other words, it should be enough to do one full pass over the plan collecting validities. When a change is made, these validities will be used to check whether a paradox occurs, and the change integrated into the validities. With this approach, it is not necessary to look at the whole plan every time it changes, only the parts that may be affected. Another benefit to this view of feature model evolutions plans is that the cause of a paradox is more easily detected than in the existing solutions; when the object of analysis is *change* to a plan and not the plan itself, the analysis becomes more specific and thus can give the user better suggestions for how to fix a paradox.

3.3 Plan for project execution

Define semantics for operations on plans

To be able to analyse the effects of a plan change, it is necessary to define change. Semantically, a change must be an operation on the plan itself. We can then use these operations and their semantics to formalize dependencies and paradoxes.

Find appropriate structure

For the analysis to be as efficient as intended, the structure of the dependencies must be optimized for near-constant lookup.

Implement a tool for static analysis based on the new semantics for feature model evolution plans.

The thesis will benefit from a proof of concept. We will implement a tool utilizing methods from static analysis and apply it on a use case.

Bibliography

[1] Domenico Bianculli et al. "Incremental Syntactic-Semantic Reliability Analysis of Evolving Structured Workflows". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 8802. Lecture Notes in Computer Science. Springer, 2014, pp. 41–55. DOI: `10.1007/978-3-662-45234-9\_4`. URL: `https://doi.org/10.1007/978-3-662-45234-9%5C_4`.

[2] Domenico Bianculli et al. "Syntax-Driven Program Verification of Matching Logic Properties". In: *3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE 2015, Florence, Italy, May 18, 2015*. Ed. by Stefania Gnesi and Nico Plat. IEEE Computer Society, 2015, pp. 68–74. DOI: `10.1109/FormaliSE.2015.18`. URL: `https://doi.org/10.1109/FormaliSE.2015.18`.

[3] Latifaah and R. Manurung. "Syntax-driven semantic analysis for constructing use case diagrams from software requirement specifications in Indonesian". In: *2012 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. 2012, pp. 149–154.

[4] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. ISBN: 978-3-540-65410-0. DOI: `10.1007/978-3-662-03811-6`. URL: `https://doi.org/10.1007/978-3-662-03811-6`.