Definitions and semantics

Ida Sandberg Motzfeldt

January 14, 2021

Contents

1 Definitions

TODO: This section assumes knowledge of normal feature models and evolution plans, and only defines the specific definitions used in this thesis.

TODO: Look at default maps

1.1 Feature model evolution plan

A feature model evolution plan is defined as a triple (NAMES, FEATURES, GROUPS) where NAMES, FEATURES, and GROUPS are all maps.

1.1.1 Maps

In general, a map is a set of entries on the form $[\, k \mapsto v \,]$. Looking up a value at the key k in map $\textsc{map}$ looks like this:

$$\textsc{map}\,[k]$$

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

$$\textsc{map}\,[k] \leftarrow v'$$

This also works if k is not already in the map. We assume $O(1)$ for lookup and insertion.

1.1.2 Intervals

To define the map values, we must first define an interval. We use the familiar mathematical notation $\left[t_{\text{start}}, t_{\text{end}}\right)$ for the intervals, where t_{start} and t_{end} denote points in time. The intervals are left-closed and right-open, meaning that t_{start} is included in the interval, and all time points until but not including t_{end}. The intervals are always used to give information about when something is true in the feature model. This will be further explained in subsections 1.1.5, 1.1.6. TODO: group map subsection.

1.1.3 Interval maps

An interval map is a map where the key is an interval (section 1.1.2). To look up values, one can either give an interval or a time point as key. Both will return sets of values. For instance, if an interval map $\textsc{i}$ contains the entry $\left[\,\left[t_1, t_5\right) \mapsto v\,\right]$, all of the queries in Figure 1 will return $\{v\}$ (assuming that $t_1 < t_2 < \ldots < t_5$):

$$\begin{gathered}
\textsc{i}\left[t_1\right] \\
\textsc{i}\left[t_3\right] \\
\textsc{i}\left[\left[t_1, t_5\right)\right] \\
\textsc{i}\left[\left[t_2, t_4\right)\right]
\end{gathered}$$

Figure 1: Interval map example

TODO: enclosingInterval function; find out which keys contain an interval

1.1.4 Interval sets

An interval set is a set of intervals with a few custom operators. Given an interval set IS, $\left[t_n, t_m\right) \in \text{IS}$ if $\left[t_n, t_m\right)$ is a member of the set, which is the expected semantics of $\in$. We define a similar operator $\dot{\in}$ such that $\left[t_n, t_m\right) \;\dot{\in}\; \text{IS}$ iff there exists some interval $[t_i, t_j) \in \text{IS}$ with $t_i \leq t_n \leq t_m \leq t_j$, i.e. an interval in IS which contains $[t_n, t_m)$. We further define the operator $\hat{\in}$ such that $[t_n, t_m) \mathrel{\hat{\in}} \text{IS}$ iff there exists some interval $[t_i, t_j) \in \text{IS}$ with $[t_n, t_m)$ overlapping $[t_i, t_j)$. Two intervals $[t_n, t_m)$ and $[t_i, t_j)$ overlap iff there exists a time point t_k with $t_n \leq t_k < t_m$ and $t_i \leq t_k < t_j$, i.e. a time point contained by both intervals.

Notice that $\in \subseteq \dot{\in} \subseteq \hat{\in}$, which means that if $[t_n, t_m) \in \text{IS}$ then also $[t_n, t_m) \mathrel{\dot{\in}} \text{IS}$, and $[t_n, t_m) \mathrel{\hat{\in}} \text{IS}$.

1.1.5 Mapping names

The NAMES map has entries of the form $[\,$**name** $\mapsto$ *interval map*$\,]$. Assuming $[$ `name` $\mapsto \text{IM}\,] \in$ NAMES, the interval map IM contains mappings on the form $[\,[t_{\text{start}}, t_{\text{end}}) \mapsto$ `featureID`$\,]$, where `featureID` is the ID of some feature in the feature model evolution plan. This should be interpreted as "*The name* `name` *belongs to the feature with ID* `featureID` *from* t_{start} *to* t_{end}". Looking up a name which does not exist will return an empty map $\emptyset$.

1.1.6 Mapping features

TODO: Find a fitting name/symbol for *feature intervals*

The FEATURES map has entries of the form $[$ `featureID` $\mapsto$ *feature intervals*$\,]$. Since several pieces of information are crucial to the analysis of a feature, it is not enough to have a simple mapping as we have for names. A feature has a name, a type, a parent group, and zero or more child groups. These can all be defined in terms of intervals (TODO: rephrase) and collected into a 5-tuple (EXISTENCE, NAMES, TYPES, PARENTGROUPS, CHILDGROUPS), where EXISTENCE is an interval set (or an interval map with constant values), NAMES is an interval map with name values, TYPES is a map with the feature's variation types, PARENTGROUPS is a map with group ID values, and childGroups is an interval map with group ID values, the interval keys possibly overlapping.

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

1.1.7 Mapping groups

TODO: Find a fitting name/symbol for *group intervals*

The GROUPS map has entries of the form $[$ `groupID` $\mapsto$ *group intervals*$\,]$. A group has a type, a parent feature, and zero or more child features. These can all be defined in terms of intervals and collected into a 4-tuple (EXISTENCE, TYPES, PARENTFEATURES, CHILDFEATURES), where EXISTENCE is an interval set (or an interval map with constant values), TYPES is a map with the group's types, PARENTFEATURES is a map with feature IDs, and CHILDFEATURES is a map with feature ID values, the interval keys possibly overlapping.

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

1.1.8 Example evolution plan

I've created a small example (see Figure 2 on the following page) containing three features and one group, describing a feature model evolution plan for a washing machine. The washing machine always has a washer, and a dryer is added at t_5. I say that the example is small, but written out it looks huge. This is because there is a lot of redundancy, which is necessary for scoping etc. TODO: rewrite this, the language is too informal

1.2 Operations

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

- **addFeature(featureID, parentGroupID, name, featureType)** from t_n to t_m
  Adds feature with id `featureID`, name `name`, and feature variation type `featureType` to the group with id `parentGroupID` in the interval $\left[t_n, t_m\right)$. `featureID` must be fresh, and the name cannot belong to any other feature in the model during the interval. The parent group must exists during the interval, and the types of the feature and the parent group must be compatible. If the feature has type **mandatory**, then the parent group must have type **AND**.

- **addGroup(groupID, parentFeatureID, groupType)** from t_n to t_m
  Adds group with id `groupID` and type `groupType` to the feature with id `parentFeatureID` during the interval [t_n, t_m *.ThegroupIDmustbefresh, andtheparentfeaturemustexistduringtheinterval*.**remove**
  Removes the feature with ID `featureID` from the feature model at t_n (does not affect possible reintroductions). The FEATURES map in the original plan must contain a mapping [`featureID` $\mapsto$ (EXISTENCE, ...)] such that $\left[t_i, t_j\right) \in$ EXISTENCE with $t_i \leq t_n \leq t_j$. The feature must not have any child groups during $\left[t_n, t_j\right)$. After removing the feature, all interval mappings should be updated from $\left[t_i, t_j\right)$ to $\left[t_i, t_n\right)$.

- **removeGroup(groupID)** at time t_n
  Removes the group with ID `groupID` from the feature model at t_n (does not affect possible reintroductions). The GROUPS map in the original plan must contain a mapping $\left[\texttt{groupID} \mapsto (\text{EXISTENCE}, \ldots)\right]$ such that $\left[t_i, t_j\right) \in$ EXISTENCE with $t_i \leq t_n \leq t_j$. The group must not have any child features during $\left[t_n, t_j\right)$. After removing the group, all interval mappings should be updated from $\left[t_i, t_j\right)$ to $\left[t_i, t_n\right)$.

- **moveFeature(featureID, targetGroupID)** at t_n
  Moves the feature with id `featureID` to the group with ID `targetGroupID`.

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

Figure 2: Small feature model evolution plan

The move cannot be done if it introduces a cycle; that is, if the target group is in the feature's subtree. The feature's subtree is moved along with it. Parent group and child feature mappings are updated accordingly.

- **moveGroup(groupID, targetFeatureID)**
  Moves the group with id `groupID` to the feature with ID `targetFeatureID`. Very similar to **moveFeature**.

- **changeFeatureVariationType(featureID, newType)**
  Changes the feature variation type of the feature with ID `featureID` to `newType`. If the new type is **mandatory**, the parent group type must be **AND**.

- **changeGroupVariationType(groupID, newType)**
  Changes the group variation type of the group with ID `groupID` to `newType`. If the new type is **OR** or **XOR**, make sure that no child feature has type **mandatory**.

- **changeFeatureName(featureID, name)**
  Changes the name of the feature with ID `featureID` to `name`. No other feature may have the same name.

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

2 Define the scope

What is the scope?

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

Time

Which timepoints of the plan may be affected by the change?

Space

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

Operation scopes

We define the *minimal* scope for each operation.

- **addFeature(featureID, parentGroupID, name, featureType)** from t_n to t_m
  When we add a feature from t_n to t_m, it is quite obvious that the scope in time will be [t$_n$, $t_m, sincethisistheintervalinwhichthefeaturewillexist.Thespatialscopemustbeonlytheparentgroup :$

If thegrouptypechangestoaconflictingone, theoperationisunsound.If theparentgroupisremoved, wehaveanorp
to t_m
The scopes are very similar in this and the preceding rule. The scope in time is $\left[t_n,\, t_m\right)$, and the scope in space is the parent feature, for which the only conflicting event is removal – the types of a group and its parent never conflict.

- **removeFeature(featureID)** at t_n
  If the original interval containing t_n in which the feature exists inside the feature model is [t$_m$, t_k , *thenthetemporalscopeis* $\left[t_n,\, t_k\right)$; since the feature is removed at t_k in the original plan, and the original plan is sound as we assume, removing the feature earlier may only affect the plan in the interval between these two time points.
  The spatial scope must be the feature's subtree. If the feature has or will have a child group during the interval, then it cannot be removed. Otherwise, there are no conflicts.

- **removeGroup(groupID)** at t_n
  Extremely similar to **removeFeature**.

- **moveFeature(featureID, targetGroupID)** at t_n
  If t_m is the time at which the feature is next moved, the temporal scope is t_n, t_m, since this operation only affects the plan within this interval.
  The spatial scope is discussed in more detail in the **move feature algo**. This scope is the largest and hardest to define, because we have to detect cycles. The scope is defined by the feature and its ancestors, as well as target group and its ancestors, which may change during the intervals. It is not necessary to look at all ancestors, only the ones which **feature** and **targetGroup** do not have in common. As usual, conflicting types and removal must be considered in addition to cycles.

- **moveGroup(groupID, targetFeatureID)** at t_n
  See moveFeature. Very similar.

- **changeFeatureVariationType(featureID, newType)** at t_n
  Temporal scope: $\left[t_n,\, t_m\right)$ if t_m is the next time point at which the feature's type changes or when feature is (next) removed.
  Spatial scope: The only possibly conflicting thing in the feature model is the parent group's type. At no point must the feature have type 'mandatory' and the parent group have type 'alternative' or 'or'. Thus, the spatial scope is the parent group.

- **changeGroupVariationType(groupID, newType)** at t_n Temporal scope: Same as previous.
  The spatial scope are the group's child features; the possible conflict is the same as with changeFeatureType.

- **changeFeatureName(featureID, name)** at t_n Temporal scope: Same as previous.

Spatial scope: The name. If it already exists within the feature model during the interval, then the change is invalid.

TODO: deal with batch operations/reverting a change: It is currently impossible to *extend* an interval; If a feature exists during $[t_3, t_5)$, it is impossible to change the plan such that it exists during $[t_3, t_6)$ instead. Don't really know how to fix that, except maybe adding an operation. In Figure 2 on page 5, if we try to change the name of feature 1 to Dryer at t_2, intending to change it back before Dryer is added, then these semantics will reject the first change, as two features will have the name Dryer during $[t_5, \infty,$ $.Theparadoxwouldberightedonceweaddthatthenamewillchangebackto Washerat\mathrm{t}_4$. There are workarounds for this, for instance changing the name of feature 2 to some temporary placeholder, making the changes to feature 1, and then changing feature 2 back. This, however, seems too cumbersome. Hopefully this use case is not common enough that most users will suffer for it, but it is definitely an example of the semantics being too strict.

3 SOS rules

3.1 Add feature rule

$$\text{(ADD-FEATURE)}$$

$$\frac{\begin{array}{c}\mathrm{FEATURES}[\texttt{fid}] = (F_e, F_n, F_t, F_p, F_c) \\ \mathrm{GROUPS}[\texttt{parentGroupID}] = (G_e, G_t, G_p, G_c) \\ [t_n, t_m) \mathbin{\widehat{\notin}} F_e \qquad [t_n, t_m) \mathbin{\dot{\in}} G_e \qquad \mathrm{NAMES}[\texttt{name}]\, [t_n, t_m) = \emptyset \\ \forall \texttt{gt}_{\in G_t[t_n, t_m)}(\texttt{compatibleTypes}(\texttt{gt}, \texttt{type}))\end{array}}{\begin{array}{c}(\mathrm{NAMES}, \mathrm{FEATURES}, \mathrm{GROUPS}) \\ : \textbf{addFeature}(\texttt{fid}, \texttt{name}, \texttt{type}, \texttt{parentGroupID}) \text{ at } [t_n, t_m) \\ \longrightarrow \\ (\mathrm{NAMES}[\texttt{name}]\, [t_n, t_m) \leftarrow \texttt{fid} \\ \mathrm{FEATURES}[\texttt{fid}] \leftarrow \texttt{setFeatureAttributes}(\mathrm{FEATURES}[\texttt{fid}], [t_n, t_m), \texttt{name}, \texttt{type}, \texttt{parentGroupID}), \\ \mathrm{GROUPS}[\texttt{parentGroupID}] \leftarrow \texttt{addChildFeature}(\mathrm{GROUPS}[\texttt{parentGroupID}], [t_n, t_m), \texttt{fid}))\end{array}}$$

Figure 3

Figure 3 describes the semantics of the **addFeature** operation. To add a feature during the interval $[t_n, t_m)$, its ID cannot exist exist during the interval ($[t_n, t_m) \mathbin{\widehat{\notin}} F_e$). The parent feature must exist ($[t_n, t_m) \mathbin{\dot{\in}} G_e$), and the types it has during the interval must be compatible with the type of the added feature ($\forall \texttt{gt}_{\in G_t[t_n, t_m)}(\texttt{compatibleTypes}(\texttt{gt}, \texttt{type}))$). The name of the feature must be available during the interval ($\mathrm{NAMES}[\texttt{name}]\, [t_n, t_m) = \emptyset$). Notice that the default value in the FEATURES map lets us treat a failed lookup as a feature, thus allowing us to express the semantics of adding a feature using only one rule.

To make the rule tidier, we use three helper functions: `compatibleTypes` (Figure 4), `setFeatureAttributes` (Figure 5), and `addChildFeature` (Figure 6).

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

Figure 4

```
setFeatureAttributes(($F_e, F_n, F_t, F_p, F_c$), $[t_{start}, t_{end})$, name, type, parentGroupID)
  = ( $F_e \cup [t_{start}, t_{end})$
    , $F_n$[$[t_{start}, t_{end})$] $\leftarrow$ name
    , $F_t$[$[t_{start}, t_{end})$] $\leftarrow$ type
    , $F_p$[$[t_{start}, t_{end})$] $\leftarrow$ parentGroupID
    , $F_c$ )
```

Figure 5

```
addChildFeature(($G_e, G_t, G_p, G_c$), $[t_{start}, t_{end})$, fid)
  = ( $G_e, G_t, G_p$
    , $G_c \cup [\, [t_{start}, t_{end}) \mapsto$ fid$\,]$ )
```

Figure 6

TODO: Find a way to express that there are no intervals in F_e overlapping $[t_n, t_m)$

Figure 7