Definitions and semantics

Ida Sandberg Motzfeldt

February 8, 2021

Contents

TODO: Explain why it's non-trivial, why it may be difficult to do it manually, why we need to restrict the scope etc. Why simple lookup is not enough.

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 ($\textsc{names}, \textsc{features}, \textsc{groups}$) where $\textsc{names}$, $\textsc{features}$, and $\textsc{groups}$ are all maps.

1.1.1 Maps

TODO: Motivate choice of data structure (mainly efficiency and modularity)

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. For maps with set values, we define an additional operator $\overset{\cup}{\longleftarrow}$. If $\textsc{map}\,[k] = S$ then

$$\textsc{map}\,[k] \overset{\cup}{\longleftarrow} v = \textsc{map}\,[k] \leftarrow S \cup \{v\}$$

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

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

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

1.1.2 Intervals

To define the map values, we must first define an interval. We use the familiar mathematical notation $[t_{\text{start}}, t_{\text{end}})$ 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 temporal feature model. This will be further explained in subsections 1.1.5, 1.1.6. Single time points t_n may be viewed as intervals containing only the time point t_n

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

1.1.3 Interval maps

An interval map is a map where the key is an interval (section 1.1.2 on the preceding page). 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 mapping $[\,[t_1, t_5) \mapsto v\,]$, all of the queries in Figure 1 will return $\{v\}$ (assuming that $t_1 < t_2 < \ldots < t_5$):

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

Figure 1: Interval map example

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

We furthermore define function IM$[t_n, t_m)_{\lesseqgtr}$ which returns all the interval keys in the map IM overlapping the interval $[t_n, t_m)$. See section 1.1.4 for definition of overlapping intervals.

1.1.4 Interval sets

An interval set is a set of intervals with a few custom predicates and operations. Given an interval set IS, $[t_n, t_m) \in$ IS if $[t_n, t_m)$ is a member of the set, which is the expected semantics of $\in$. We define a similar predicate $\in_{\leq}$ such that $[t_n, t_m) \in_{\leq}$ IS iff there exists some interval $[t_i, t_j) \in$ IS with $t_i \leq t_n \leq t_m \leq t_j$, i.e. an interval in IS which contains $[t_n, t_m)$. We further define the predicate $\in_{\lesseqgtr}$ such that $[t_n, t_m) \in_{\lesseqgtr}$ IS iff there exists some interval $[t_i, t_j) \in$ 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 $\in \subseteq \in_{\leq} \subseteq \in_{\lesseqgtr}$, which means that if $[t_n, t_m) \in$ IS then also $[t_n, t_m) \in_{\leq}$ IS, and $[t_n, t_m) \in_{\lesseqgtr}$ IS.

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

1.1.5 Mapping names

The $\textsc{names}$ map has entries of the form $[\,$**name** $\mapsto$ *interval map*$\,]$. Assuming $[\,$`name` $\mapsto \textsc{Im}\,] \in \textsc{names}$, the interval map $\textsc{Im}$ 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: Child features/groups cannot have identical keys, change semantics

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

The root ID is constant for a temporal feature model. We assume that it has been computed and is stored in the variable `rootID`. In a balanced feature model, locating the root is $O(\log n)$ if we start at an arbitrary feature and follow the ancestor chain.

1.1.7 Mapping groups

The GROUPS map has entries of the form $[$`groupID` $\mapsto$ *group entry*$]$. A group has a type, a parent feature, and zero or more child features. It may also be removed and re-added These can all be defined in terms of intervals and collected into a 4-tuple similarly to the feature entries (EXISTENCE, TYPES, PARENTFEATURES, CHILDFEATURES), where EXISTENCE is an interval set denoting when the group exists, 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 page 6) 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 $[t_n, t_m)$. `featureID` must be fresh, and the name cannot belong to any other feature in the model during the interval. The parent group must 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 $[t_n, t_m)$. The group ID must be fresh, and the parent feature must exist during the interval.

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

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

- **moveFeature(featureID, targetGroupID)** at t_n
  Moves the feature with id `featureID` to the group with ID `targetGroupID`. The move cannot be done if it introduces a cycle; that is, if the target group

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

Figure 2: Small feature model evolution plan

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
  TODO: Give an argument why the parent group is the spatial scope of this rule When we add a feature from t_n to t_m, it is quite obvious that the scope in time will be $[t_n, t_m)$, since this is the interval in which the feature will exist. The spatial scope must be only the parent group: If the group type changes to a conflicting one, the operation is unsound. If the parent group is removed, we have an orphaned feature, which is also illegal. But what about the name? If we only stored name information inside features, we would have to check every feature in the whole interval for this change. However, since we store information about names in a separate map, we can look up the name and check that it is not in use during the interval. TODO: Consider moving last part to data structure definitions

- **addGroup**(`groupID`, `parentFeatureID`, `groupType`) from t_n to t_m
  The scopes are very similar in this and the preceding rule. The scope in time is $[t_n, t_m)$, and the scope in space is the 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
  TODO: Explain why the temporal scope is the way it is If the original interval containing t_n in which the feature exists inside the feature model is $[t_m, t_k)$, then the temporal scope is $[t_n, t_k)$ - from the feature is removed until it would have been removed anyway TODO: rephrase. Since the feature is removed at t_k in the original plan, and the original plan is sound as we assume, removing the feature earlier may only affect the plan in the interval between these two time points.
  The spatial scope must be the feature'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 in the original plan, the temporal scope is $[t_n, t_m)$, since this operation only affects the plan within this interval.
  The spatial scope is discussed in more detail in the **move feature algo**. This scope is the largest and hardest to define, because we have to detect cycles. The scope is defined by the feature and its ancestors, as well as target group and its ancestors, which may change during the intervals. It is not necessary to look at all ancestors, only the ones which `feature` and `targetGroup` do not have in common. As usual, conflicting types and removal must be considered in addition to cycles.

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

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

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

- **changeFeatureName**(`featureID`, `name`) at t_n Temporal scope: Same as previous.
  Spatial scope: The name. 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 6, if we try to change the name of feature 1 to Dryer at t_2, intending to change it back before Dryer is added, then these semantics will reject the first change, as two features will have the name Dryer during $[t_5, \infty)$. The paradox would be righted once we add that the name will change back to Washer at t_4. There are workarounds for this, for instance changing the name of feature 2 to some temporary placeholder, making the changes to feature 1, and then changing feature 2 back. This, however, seems too cumbersome. Hopefully this use case is not common enough that most users will suffer for it, but it is definitely an example of the semantics being too strict.

3 SOS rules

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

TODO: Remember premises = above the line

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

SOS rules TODO: explain what SOS rules are used for

The rules are on the form

(Rule-Label)

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

where S is the state, and S' is the new state after the rule is applied. The rule can only be applied if all the premises hold. In this thesis, the state is always on the form **operation** $\triangleright$ $(\textsc{names}, \textsc{features}, \textsc{groups})$, where **operation** denotes the change we intend to make to the temporal feature model $(\textsc{names}, \textsc{features}, \textsc{groups})$. The new state is always on the form $(\textsc{names}', \textsc{features}', \textsc{groups}')$, where the maps have been updated according to the semantics of the operation. The premises ensure that an operation can only be applied if some conditions hold; for instance the **Add-Feature** rule 3 contains premises verifying that the feature does not already exist when we wish to add it.

3.1 Add feature rule

(Add-Feature)

$$\frac{\begin{array}{c} [t_n, t_m) \not\in_{\lneqq} F_e \qquad [t_n, t_m) \in_{\leq} G_e \qquad \textsc{names}\,[\texttt{name}]\,[t_n, t_m) = \emptyset \\ \textsc{features}\,[\texttt{fid}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c) \\ \textsc{groups}\,[\texttt{parentGroupID}] = (G_e,\, G_t,\, G_p,\, G_c) \\ \forall \texttt{gt}_{\in G_t[t_n, t_m)}(\texttt{compatibleTypes}(\texttt{gt}, \texttt{type})) \end{array}}{\begin{array}{c} \textbf{addFeature}(\texttt{fid}, \texttt{name}, \texttt{type}, \texttt{parentGroupID}) \text{ at } [t_n, t_m) \triangleright \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ (\textsc{names}\,[\texttt{name}]\,[t_n, t_m) \leftarrow \texttt{fid}, \\ \textsc{features}\,[\texttt{fid}] \leftarrow \texttt{setFeatureAttributes}(\textsc{features}\,[\texttt{fid}]\,, [t_n, t_m), \texttt{name}, \texttt{type}, \texttt{parentGroupID}), \\ \textsc{groups}\,[\texttt{parentGroupID}] \leftarrow \texttt{addChildFeature}(\textsc{groups}\,[\texttt{parentGroupID}]\,, [t_n, t_m), \texttt{fid})) \end{array}}$$

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 $([t_n, t_m) \not\in_{\lneqq} F_e)$. The parent feature must exist $([t_n, t_m) \in_{\leq} G_e)$, and the types it has during the interval must be compatible with the type of the added feature $(\forall \texttt{gt}_{\in G_t[t_n, t_m)}(\texttt{compatibleTypes}(\texttt{gt}, \texttt{type})))$. The name of the feature must not be in use during the interval $(\textsc{names}\,[\texttt{name}]\,[t_n, t_m) = \emptyset)$. Notice that the default value in the $\textsc{features}$ 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

3.2 Add group rule

The rule in figure 7 describes the conditions which must be in place to add a (pre-existing or fresh) group to the FMEP during an interval $[t_n, t_m)$. The group must not already exist in the plan during the interval $[t_n, t_m) \notin_{\lesseqgtr} G_e$, and the parent feature must exist for the duration of the interval $[t_n, t_m) \in_{\leq} F_e$. The group ID is added to the parent feature's map of child groups with the interval as key, and the attributes specified are added to the group entry in the GROUPS map.

3.3 Remove feature rule

Figure 10 shows the semantics of removing a feature with ID `featureID` at time t_n. We find the time point when the feature was to be removed in the original plan by looking up the interval containing t_n in the feature's EXISTENCE set $[t_{e_1}, t_{e_2})$. The interval in which the new plan is different from the original is then $[t_n, t_{e_2})$. We verify that the feature does not have any child groups during the affected interval ($F_c[t_n, t_{e_2}) = \emptyset$). We furthermore check that the feature has only a single name, type, and parent during the interval. This means that the original plan did not change the feature's name, type, or parent during this time. If these conditions all hold, we update the temporal feature model by clamping all the relevant intervals to t_n, i.e. shortening them to end at t_n.

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

Figure 6

(Add-Group)

$$\frac{[t_n, t_m) \notin_{\gtreqless} G_e \qquad [t_n, t_m) \in_{\leq} F_e \\ \textsc{groups}\,[\texttt{groupID}] = (G_e,\, G_t,\, G_p,\, G_c) \\ \textsc{features}\,[\texttt{parentFeatureID}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c)}{\begin{array}{c}\textbf{addGroup}(\texttt{groupID}, \texttt{type}, \texttt{parentFeatureID}) \text{ at } [t_n, t_m) \triangleright \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ (\textsc{names}, \\ \textsc{features}\,[\texttt{parentFeatureID}] \leftarrow (F_e,\, F_n,\, F_t,\, F_p,\, F_c \cup [\,[t_n, t_m) \mapsto \texttt{groupID}\,]), \\ \textsc{groups}\,[\texttt{groupID}] \leftarrow \texttt{setGroupAttributes}(\textsc{groups}\,[\texttt{groupID}]\,, \texttt{type}, \texttt{parentFeatureID}))\end{array}}$$

Figure 7

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

Figure 8

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

Figure 9

(Remove-Feature)

$$\frac{F_e\,[t_n]_{\leq} = \{[t_{e_1}, t_{e_2})\} \qquad F_c[t_n, t_{e_2}) = \emptyset \\ F_n[t_n, t_{e_2}) = \{\texttt{name}\} \qquad F_t[t_n, t_{e_2}) = \{\texttt{type}\} \qquad F_p[t_n, t_{e_2}) = \{\texttt{parentGoupID}\} \\ \textsc{features}\,[\texttt{featureID}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c) \\ \textsc{groups}\,[\texttt{parentGroupID}] = (G_e,\, G_t,\, G_p,\, G_c)}{\begin{array}{c}\textbf{removeFeature}\,(\texttt{featureID}) \text{ at } t_n \triangleright \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ \big(\textsc{names}\,[\texttt{name}] \leftarrow \texttt{clampInterval}(\textsc{names}\,[\texttt{name}]\,, t_n), \\ \textsc{features}\,[\texttt{featureID}] \leftarrow \texttt{clampFeature}(\textsc{features}\,[\texttt{featureID}]\,, t_n), \\ \textsc{groups}\,[\texttt{parentGroupID}] \leftarrow (G_e,\, G_t,\, G_p,\, \texttt{clampIntervalValue}(G_c, t_n, \texttt{featureID}))\,\big)\end{array}}$$

Figure 10

```
clampInterval($\textsc{map}$, $t_c$)
  = let $\{[t_{start}, t_{end})\} \leftarrow \textsc{map}\,[t_c]_{\leq}$
        $\{v\} \leftarrow \textsc{map}\,[t_c]$
        $\textsc{map}' \leftarrow \textsc{map} \setminus [t_{start}, t_{end})$
     in $\textsc{map}'[t_{start}, t_c) \leftarrow v$
```

Figure 11

```
clampIntervalValue($\textsc{map}$, $t_c$, $v$)
  = let $\{[t_{start}, t_{end})\} \leftarrow \textsc{map}\,[t_c]^{v}_{\leq}$
        $\textsc{map}' \leftarrow \textsc{map} \setminus^{v} [t_{start}, t_{end})$
     in $\textsc{map}'[t_{start}, t_c) \overset{\cup}{\leftarrow} v$
```

Figure 12

```
clampSetInterval(IS, $t_c$)
  = let $\{[t_{start}, t_{end})\} \leftarrow \mathrm{IS}\,[t_c]_{\leq}$
        $\mathrm{IS}' \leftarrow \mathrm{IS} \setminus [t_{start}, t_{end})$
     in $\mathrm{IS}' \cup \{[t_{start}, t_c)\}$
```

Figure 13

```
clampFeature(($F_e, F_n, F_t, F_p, F_c$), $t_c$)
  = (clampSetInterval($F_e, t_c$)
    , clampInterval($F_n, t_c$)
    , clampInterval($F_t, t_c$)
    , clampInterval($F_p, t_c$)
    , $F_c$)
```

Figure 14

```
clampGroup(($G_e, G_t, G_p, G_c$), $t_c$)
  = (clampSetInterval($G_e$)
    , clampInterval($G_t, t_c$)
    , clampInterval($G_p, t_c$)
    , $G_c$)
```

Figure 15

3.4 Remove group rule

The **Remove-Group** rule in figure 16 describes the semantics of removing a group in a temporal feature model. The temporal scope is identified as the existence interval containing the time point for removal. In that interval, the group may not have any children, and there cannot be plans to change the type or move the group within the interval. We check the latter by looking up the type and parent feature during the interval; if the set contains only one type/parent feature then the type and parent feature do not change.

We use the `clampInterval` (figure 11 on the previous page), `clampIntervalValue` (figure 12 on the preceding page), and `clampGroup` (figure 15 on the previous page) helper functions to update the temporal feature model. The `clampInterval` function takes an interval map with non-overlapping keys and a time point t_c, and updates the interval key containing t_c to end at t_c. `clampIntervalValue` does the same, but for interval maps with overlapping keys and set values. It takes an interval map, a time point t_c, and a value v, and shortens the interval key containing t_c and v to end at t_c. `clampSetInterval` takes an interval set with non-overlapping values and a time point t_c, and shortens the interval containing t_c.

(Remove-Group)

$$\frac{\begin{array}{c} G_e\,[t_n]_{\leq} = \{[t_{e_1}, t_{e_2})\} \qquad G_c[t_n, t_{e_2}) = \emptyset \\ G_t[t_n, t_{e_2}) = \{\texttt{type}\} \qquad G_p[t_n, t_{e_2}) = \{\texttt{parentFeatureID}\} \\ \textsc{groups}\,[\texttt{groupID}] = (G_e,\, G_t,\, G_p,\, G_c) \\ \textsc{features}\,[\texttt{parentFeatureID}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c) \end{array}}{\begin{array}{c} \textbf{removeGroup}\,(\texttt{groupID})\ \text{ at } t_n \triangleright \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ \big(\textsc{names}\,[\texttt{name}] \leftarrow \texttt{clampInterval}(\textsc{names}\,[\texttt{name}]\,, t_n), \\ \textsc{features}\,[\texttt{parentFeatureID}] \leftarrow (F_e,\, F_n,\, F_t,\, F_p,\, \texttt{clampIntervalValue}(F_c, t_n, \texttt{groupID}))\,, \\ \textsc{groups}\,[\texttt{groupID}] \leftarrow \texttt{clampGroup}(\textsc{groups}\,[\texttt{groupID}]\,, t_n)\big) \end{array}}$$

Figure 16

3.5 Move feature rule

The `ancestors` function (Figure 17 on the following page) returns a list of ancestors for a feature ID *or* a group ID.

3.6 Move group rule

TODO: This is hard

```
ancestors($(\textsc{names}, \textsc{features}, \textsc{groups})$, featureID, $t_n$)
  = let  $(F_e, F_n, F_t, F_p, F_c) \leftarrow \textsc{features}\,[$featureID$]$
         parentGroup $\leftarrow F_p\,[t_n]$
      in
       case parentGroup of
         { parentGroupID } $\rightarrow$
           parentGroupID : ancestors($(\textsc{names}, \textsc{features}, \textsc{groups})$, parentGroupID, $t_n$)
         $\emptyset \rightarrow$ []
ancestors($(\textsc{names}, \textsc{features}, \textsc{groups})$, groupID, $t_n$)
  = let  $(G_e, G_t, G_p, G_c) \leftarrow \textsc{groups}\,[$groupID$]$
         { parentFeatureID } $\leftarrow G_p\,[t_n]$
      in
         parentFeatureID : ancestors($(\textsc{names}, \textsc{features}, groups)$, parentFeatureID, $t_n$)
```

Figure 17

```
hasCycles($(\textsc{names}, \textsc{features}, \textsc{groups}), n, c, [t_{start}, t_{end})$)
  = let  $A_n \leftarrow$ ancestors($n, t_{start}$)
         $A_c \leftarrow$ ancestors($c, t_{start}$)
         $C \leftarrow c$ : -- critical nodes (A_c cut off at the first node in common with A_n)
      in
       if $n \in A_c$ then True
       else
         let r $\leftarrow$ firstMove($C$)
          in if r =
```

Figure 18

3.7 Change feature variation type rule

The rule in figure 19 on the next page shows the semantics of changing the feature variation type of the feature with ID `featureID` at time t_n. The first expression above the line ($F_t\,[t_n]_{\leq} = \{[t_{t_1}, t_{t_2})\}$) identifies the upper bound of the temporal scope, t_{t_2}. This is when the feature type was originally planned to change. The next line may be hard to read, but its intent is easier to understand. It checks that all the types a parent group has *while it is the parent of the feature* has a type which is compatible with the new type of the feature. If everything above the line is true, then the FEATURES map is updated at `featureID` by shortening the interval key for the original type at t_n, and assigning the new type to the affected interval $[t_n, t_{t_2})$.

TODO: make it more readable

(Change-Feature-Variation-Type)

$$\frac{\begin{array}{c} F_t\,[t_n]_{\le} = \{[t_{t_1}, t_{t_2})\} \\ \forall [t_{p_1}, t_{p_2}) \in F_p[t_n, t_{t_2})_{\lesseqgtr} \forall p \in F_p[t_{p_1}, t_{p_2}) \forall t \in \texttt{getTypes}\left(\textsc{groups}\,[p]\,, \langle [t_{p_1}, t_{p_2}) \rangle_{t_n}^{t_{t_2}}\right) \cdot \left(\texttt{compatibleTypes}(t, \texttt{type})\right) \\ \textsc{features}\,[\texttt{featureID}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c) \end{array}}{\begin{array}{c} \textbf{changeFeatureVariationType}\,(\texttt{featureID}, \texttt{type}) \;\text{ at } t_n \triangleright \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ (\textsc{names}, \\ \textsc{features}\,[\texttt{featureID}] \leftarrow (F_e,\, F_n,\, \texttt{clampInterval}(F_t, t_n)[t_n, t_{t_2}) \leftarrow \texttt{type},\, F_p,\, F_c)\,, \\ \textsc{groups}) \end{array}}$$

Figure 19

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

Figure 20

3.8 Change group variation type rule

The rule in figure 21 is similar to the **changeFeatureVariationType** rule in figure 19, and shows the semantics of changing the type of a group. In a similar way to the aforementioned **changeFeatureVariationType** rule, it verifies that the types of all the child groups during the affected interval are compatible with the new group type.

3.9 Change feature name

The semantics of changing the name of a feature are shown in the **Change-Feature-Name** rule in figure 22 on the following page. The old name and the

(Change-Group-Variation-Type)

$$\frac{\begin{array}{c} G_t\,[t_n]_{\le} = \{[t_{t_1}, t_{t_2})\} \\ \forall [t_{c_1}, t_{c_2}) \in G_c[t_n, t_{t_2})_{\lesseqgtr} \forall c \in \bigcup G_c[t_{c_1}, t_{c_2}) \forall t \in \texttt{getTypes}\left(\textsc{features}\,[c]\,, \langle [t_{c_1}, t_{c_2}) \rangle_{t_n}^{t_{t_2}}\right) \cdot \left(\texttt{compatibleTypes}(\texttt{type}, t)\right) \\ \textsc{groups}\,[\texttt{groupID}] = (G_e,\, G_t,\, G_p,\, G_c) \end{array}}{\begin{array}{c} \textbf{changeGroupVariationType}\,(\texttt{groupID}, \texttt{type}) \;\text{ at } t_n \triangleright \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ (\textsc{names}, \textsc{features}, \\ \textsc{groups}\,[\texttt{groupID}] \leftarrow (G_e,\, \texttt{clampInterval}(G_t, t_n)[t_n, t_{t_2}) \leftarrow \texttt{type},\, G_p,\, G_c)) \end{array}}$$

Figure 21

(Change-Feature-Name)

$$\frac{\begin{array}{c} F_n\,[t_n] = \{\texttt{oldName}\} \qquad F_n\,[t_n]_{\leq} = \{[t_{n_1}, t_{n_2})\} \\ \textsc{names}\,[\texttt{name}]\,[t_n, t_{n_2}) = \emptyset \\ \textsc{features}\,[\texttt{featureID}] = (F_e,\, F_n,\, F_t,\, F_p,\, F_c) \end{array}}{\begin{array}{c} \textbf{changeFeatureName}\,(\texttt{featureID}, \texttt{name}) \;\text{ at } t_n \triangleright \\ (\textsc{names}, \textsc{features}, \textsc{groups}) \\ \longrightarrow \\ ((\textsc{names}\,[\texttt{name}]\,[t_n, t_{n_2}) \leftarrow \texttt{featureID})\,[\texttt{oldName}] \leftarrow \texttt{clampInterval}(\textsc{names}\,[\texttt{oldName}]\,, t_n), \\ \textsc{features}\,[\texttt{featureID}] \leftarrow (F_e,\, \texttt{clampInterval}(F_n, t_n)[t_n, t_{n_2}) \leftarrow \texttt{name},\, F_t,\, F_p,\, F_c)\,, \\ \textsc{groups}) \end{array}}$$

Figure 22

next planned name change are identified on the first line ($F_n\,[t_n] = \{\texttt{oldName}\}$ and $F_n\,[t_n]_{\leq} = \{[t_{n_1}, t_{n_2})\}$ respectively). Since the name must not be in use during the temporal scope, we verify that looking up the new name in the NAMES map returns an empty set. The NAMES map is updated by shortening the interval for the old name to end at t_n, and assigning the feature ID to the new name during the temporal scope. Furthermore, the FEATURES map is updated at the feature ID, shortening the interval for the old name and assigning the new name to the temporal scope.