Supporting Multi-level Configuration with Feature-Solution Graphs
Formal Semantics and Alloy Implementation

Jaime Chavarriaga∗1,2, Carlos Noguera†1, Rubby Casallas‡2, and Viviane Jonkers§1

1Vrije Universiteit Brussel
2Universidad de los Andes

Abstract

In Software Product Lines, several approaches support configuration processes using feature models representing constraints about which features, software elements or assets can be included in a product. These approaches use a single feature models to display users which features can be included in a product and to validate if a given configuration is valid. In this paper we propose feature-solution graphs (FSGs) to detect and explain conflicts in multi-staged configuration processes. In these FSGs, configuration options are arranged into a pair of feature models with relationships between them. Then, when some options are selected in one of them, an automatic process is able to determine the set of options that can be selected in the other. Finally, if a combination of features results in that none option can be selected in the latter feature model, an automatic process determine the selected options that cause the problem. The technical report presents the approach providing a formal semantics for feature models and feature solution graphs, and provides an example and comparison.

1 Introduction

In Software Product Lines (SPL), building correct configurations is essential to define products that not only satisfy the requirements, but that are also

∗jchavarr@vub.ac.be
†cnoguera@vub.ac.be
‡rcasalla@uniandes.edu.co
§vejoncke@vub.ac.be
amenable to composition; i.e., configurations that do not have conflicts or in-
compatibilities. Usually the configuration of these products may require the
involvement of different stakeholders [5], and selections performed by one stake-
holder may conflict with choices of others.

Feature models have been used as a foundation for configuration processes [5, 4, 8]. Originally focused on documenting commonalities and variabilities, feature
models allow engineers to define constraints over which combinations of features
can be part of a product [11].

Multi-staged configuration [5, 8] process considers diverse stakeholders that
make decisions over the same set of features, while multi-level configuration [5]
considers a different set of features for each stakeholder. In both, the selections
performed by any stakeholder can restrict the options that the others can select.
How these processes are implemented can vary from approach to approach: some
propagate constraints of already made decisions to indicate which additional
features are or cannot be selected [1]. Other approaches [5] use the selections
performed to modify the structure and constrains in the feature models and
present to users only the subset of possibilities that they can select.

Existing multi-staged and multi-level configuration approaches focus on pre-
venting errors during configuration: selecting features in one stage/level that
cannot exist in the same configuration with features selected in subsequent
stages/levels. However, in configuration processes with multiple stakeholders,
these approaches can not explain, for an stakeholder, which features selected
by other stakeholders cause that some feature must be selected, that a feature
cannot be selected, or the causes of a conflict.

This paper presents FaMoSA (Feature Models for Software Architecture) a
multi-level configuration process based on independent feature models, to repre-
sent variability relevant to diverse stakeholders, and feature-solution graphs (FS-
Graphs) to relate features involved in one configuration level with the features
involved in the next one. We are interested on explaining to each stakeholder
why some features are selected or which decisions taken by other stakeholders
cause conflicts. Conflicting features are detected by reasoning over the struc-
ture of the FM, while their explanation is derived from the FS-Graph’s relations.
The major contributions of this paper are:

1. A formal semantics for feature models that tolerate inconsistencies during
   a configuration process.
2. A formal semantics for feature-solution graphs to describe configuration
   processes.
3. An approach to detect conflicts in a configuration process, and explain
   their causes based on transformations over feature-solution graphs.

The remainder of this paper is structured as follows. Section 2 gives an
overview of related work and 3 presents our approach for supporting multi-level
configuration processes. First, Section 3.1 presents an overview, then section
3.2 presents the FaMoSA formalization, and finally, section 3.3 presents an
implementation of the formalization in Alloy. Section 4 concludes the paper and outlines our future work.

2 Related Work

Feature-based Configuration Software Product Lines and Configurable Software, such as applications servers and operating systems, require a Configuration Process [5, 8]. In these processes, one or more stakeholders configure a product by selecting features and assigning values to options. The expected outcome is a valid configuration that includes only features and options that do not have conflicts or incompatibilities among them.

In configuration processes with multiple stakeholders, each stakeholder configures features of her concern [8]. However, selections performed by one may affect which features can be selected by the others. Several approaches have been proposed to support configuration processes and help stakeholders produce valid configurations [5, 4, 8]. Feature-based Configuration Processes are those approaches based on feature models to specify the features and options that can be selected, and the constraints about which features can be chosen at the same time [8]. Depending on the approach, they organize the options for each stakeholders using a single feature model or using different feature models for each stakeholder [5].

Multi-stage configuration processes [5, 4, 8] use a single feature model to represent the whole variability independently of the stakeholders. These approaches combine the feature models of all the stakeholders in a single model and introduce new implies and excludes relationships between features of a stakeholder and features of others. These types of relationships allow to specify that a feature must be or not included in a configuration as an effect of a selection. However, these relationships cannot be used to represent that a selection of a feature causes specific modifications on the feature model, e.g., they cannot be used to denote that, after a feature is selected, other feature becomes mandatory in the group where it belongs.

Multi-level configuration uses multiple feature models, basically, one per stakeholder. Czarnecki, Classen et al. [5, 4] use annotations in features to include expressions that determine if a feature must be included in a configuration. These expressions are defined in terms of features of other stakeholders. Bruin, Janota et al. [3, 10] propose Feature-Solution and Feature-Component Graphs to represent, after a selection of a feature, which other features in other feature models must be also included in a product configuration. However these approaches only define the equivalent to implies relationships, but not to represent that a feature prohibits another feature or makes it mandatory in the feature group where it belongs.

Conflict Tolerance during Configuration Configuration conflicts occur when intended features of a stakeholder cannot be selected because of other choices by same or other stakeholders [8].
Automatic support for detecting and preventing conflicts in multi-staged and multi-level configuration are mostly based on propagating decisions. Propagation of decisions is a process that takes the features selected by one stakeholder to update the features belonging to the same and other stakeholders to represent all the effects of these decisions.

To prevent conflicts, approaches propagate decisions using feature model specialization, i.e. reducing the variability in feature models [5]. These approaches update constantly the structure of feature models taking the features selected by the stakeholders and the constraints defined in the feature model to set mandatory those features that must be selected and to remove non-selectable features [12, 4]. They prevent configuration conflicts because conflicting features are removed. However, because missing features, the structure of the feature model cannot be used to explain what previous choices caused a conflict.

On the other hand, other approaches tolerate conflicts detecting them after stakeholders have selected all intended features. They convert feature models and partial configurations into a set of constraints and use a solver to determine if there are conflicts [2]. However, to obtain the information of already selected or non-selectable features provided by the specialization process, these approaches require further processing using a solver with additional constraints for each feature.

Configuration Conflict Explanation In configuration processes with multiple stakeholders, determining the causes of a conflict, i.e., why a feature is non-selectable or already selected, is key. If the causes of a conflict are in the features belonging to the same stakeholder, she can fix it changing her own choices. However, when causes include features selected by other stakeholders, negotiation is required to solve the conflict. This negotiation not only requires knowledge about which combinations are valid, but also about which selections of each stakeholder are the cause of the conflict and negotiate with that stakeholders which selections maintain or retract.

As mentioned before, feature model specialization alters the models removing non-selectable features. Then, the remaining structure cannot be used to explain why a feature is not selectable. When the conflicts lead to no possible valid configuration, all the features in the model are removed, making harder to explain any cause.

The approaches that tolerate conflicts use combinatorial analysis and solver techniques to determine configuration conflicts [2, 13]. However, they present information about combinations of features that result in conflicts but not about the other features involved, e.g. the features where it occurs.

In order to support solution to configuration conflicts involving multiple stakeholders, it is important to present the impact of each selection and explain causes of selected or non-selectable features in terms of features belonging to other stakeholders. This cannot be easily obtained using feature model specialization because information is lost in the process, neither using existing solver approaches focused on valid combinations of features.
3 FaMoSA

Section 3.1 presents an overview where informally, our multi-level configuration process is explained and then, section 3.2 presents the main elements of the FaMoSA formalization.

3.1 Overview

3.1.1 Feature-based Configuration

One of the contributions of our work is the FaMoSA’s ability to explain conflicts in terms of selected features of other stakeholders. In our strategy, we specialize feature models tolerating conflicts during the process, and maintaining structures that allow us to trace the decisions and explain the reasons of conflicts.

Conflict-tolerant feature models To tolerate conflicts, we propose feature models where features can be typed as non-selectable, in addition to the standard mandatory and optional types. Our feature models allow features to carry more than one type at the same time. Thus, a feature model can be well-formed even when a feature is mandatory and non-selectable. Notice that normally, a feature model would not be constructed using non-selectable features, rather the type is used to make explicit the cases in which a feature cannot be selected due to constraints in the model propagated during the configuration process.

In our approach, we distinguish full mandatory features as those that must be included in all the configurations (i.e. the mandatory features which ancestors are also mandatory features). Notice that a feature model is invalid\(^1\) if any full mandatory feature is also non-selectable.

Figure 1: Example feature model supporting multiple types. A cross-box is used to indicate non-selectable feature.

Figure 1 shows a feature model, result of a configuration step, supporting multiple types and non-selectable features. It includes an optional feature \(f_1\) and a mandatory feature \(f_2\) as traditional feature models. These features are in an Or Group denoting that both can be included in a valid configuration. In

\(^1\)If there are no valid configurations for a feature model.
addition, the model includes a feature $f_3$ that has an optional and an additional mandatory type (i.e. effectively rendering the feature mandatory). The feature $f_4$ is optional and non-selectable (i.e. it is non-selectable or a dead feature) and feature $f_5$ is optional, mandatory and non-selectable (i.e. a conflicting feature that makes its parent non-selectable). The feature $f_2$ is *full mandatory* because it and all its ancestor features are mandatory. Features $f_3$, $f_4$ and $f_5$ are in an *Alternative Group*, i.e. only one of them can be included in a valid configuration. Notice that the presented feature model is valid because no *full mandatory* feature is also *non-mandatory* (e.g. a configuration including root, $f_2$ and all its ancestors is a valid configuration).

**Feature-Solution Graphs (FS-Graphs)** We specify features in a configuration process using feature models and feature solution graphs (FS-Graphs) that relate two feature models. Instead using a single feature model, we define a different feature model for each level. We define relationships from features of one level to features in the next level; when a feature in the left-side model is selected, using the relationships in the FS-Graph, we can affect features in the right-side model.

As we explain in the related work, *implies* and *excludes* relationships are not enough to constraints features from different levels, then, we have defined three types of relationships that reflect specialization operations for the defined feature models. For a given configuration $c$ and $f$ and $f'$ features of the left and right hand side feature models of an FS-Graph respectively:

- **forces** A relation $f \xrightarrow{\text{forces}} f'$ denotes that the feature $f'$ must be converted to full-mandatory when $f$ is included in the configuration $c$.

- **suggests** A relation $f \xrightarrow{\text{suggests}} f'$ indicates that the feature $f'$ must be typed as mandatory just in the feature group where it belongs when $f$ is included in $c$.

- **prohibits** A relation $f \xrightarrow{\text{prohibits}} f'$ denotes that the feature $f'$ must be typed as a non selectable feature when $f$ is included in $c$.

**3.1.2 Configuration process**

During a configuration process, stakeholders make decisions selecting features to include in a configuration, and an automatic process propagates that decisions selecting and making non-selectable other features. This propagation of decisions is achieved through a *feature model specialization*, a transformation that takes a feature model and yields another feature such that the set of valid configurations of the resulting model is a subset of the valid configurations of the original feature model. Informally, an FS-Graph holds the information to perform a feature model specialization. Basically, the specialization process takes the features of a valid configuration in the left-side feature model, obtains the relationships defined in the FS-Graph that start on these features, and modify the types of the features in the right-side model where those relationships end.
Figure 2: Effect of *forces*, *suggests* and *prohibits* relationships in a feature model specialization.

Figure 2 shows, for each type of relationship in a *FS-Graph*, how the right-side feature model is specialized according to the semantics that we have associated to the relationships.

**forces** Figure 2.a shows an FS-Graph where a feature in the left-side has a *forces* relationship to a feature in the right-side. Then, if that feature is selected in the left-side, the corresponding feature in the right-side is converted into a *full-mandatory* feature, adding a *mandatory* type to that feature and all its ancestors.

**suggests** Figure 2.b shows a *suggests* relationship: if the feature in the right-side is selected, a *mandatory* type is added to the corresponding feature in the left-side (but not to its ancestors).

**prohibits** Figure 2.c shows a *prohibits* relationship: if the feature in the right-side is selected, a *non-selectable* type is added to the corresponding feature in the left-side.

Our *propagation of decisions* makes explicit all *mandatory* and *non-selectable* features in the right-side feature model, either because: 1) they are originally marked or, 2) they are a result of the *forces*, *suggests* and *prohibits* relationships in the FS-graphs, or 3) they are a result of the constraints defined in the right-side feature model.
In addition, this propagation creates new relationships in the FS-Graph representing, for each selected feature in the left-side model, which additional features in the right-side model are converted to mandatory or non-selectable in consequence. These updated graph maintain a trace that is later used to explain conflicts.

Figure 3: Propagation of decisions into new relationships in the feature-solution graph.

Figure 3 shows how a propagation of decisions updates an FS-Graph. The left side shows an FS-Graph where a feature in the right-side model has a forces relationships to a feature in the left-side model. After propagation, new relationships are added into the FS-Graph to denote, for the original feature in the first model, which additional features are suggested and prohibited.

Figure 4: Example FS-Graph

As a more complete example, Figure 4 presents two feature models and FS-Graphs relating features of the left-side model with features of the right-side model. These relationships specify that feature $f_1$ in the left-side forces the feature $f'_{10}$ in the right side, the feature $f_2$ forces $f'_5$ and $f_3$ prohibits $f'_2$.

According to these relationships, each time a stakeholder selects one of the features $f_1, f_2$ and $f_3$, variability in the right-side feature model is modified by adding new types to the features in that model. In addition, new relationships are added to the FS-Graphs to represent which selection caused that new types. Figure 5 shows the updated FS-Graphs that result from selecting each feature in the left-side model.

As we already mentioned, propagating the selection of a feature results in a set of changes on the right-side feature model. For instance, figure 5.a shows the propagation of selecting $f_1$ assuming that $f_1$ forces $f'_{10}$: First, $f'_{10}$ must become in a full mandatory feature, thus the features $f'_{10}$ and its ancestors, i.e. the feature $f''_6$, must be marked as mandatory. Then, because each of those
features now marked as mandatory are part of alternative groups, other features in that group must be typed as non-selectable, i.e. the features \( f'_7 \) and \( f'_9 \). In addition, because \( f'_3 \) implies \( f'_5 \) (a non-selectable) feature, \( f'_3 \) becomes non-selectable too. Finally, because \( f'_3 \) is non-selectable and it is in an alternative group with only other feature, that feature \( f'_2 \) must be typed as mandatory.

Besides the new types in the right-side feature model, new relationships are added to the FS-Graph to denote that \( f'_1 \) suggests \( f'_2 \), \( f'_8 \) and \( f'_9 \), and prohibits \( f'_3 \), \( f'_7 \) and \( f'_9 \).

Figure 5.b shows that propagating the decision of selecting \( f_2 \), considering that \( f_2 \) forces \( f'_5 \), \( f'_4 \) and \( f'_2 \). Figure 5.c shows that propagating the decision of selecting of \( f_3 \), considering that \( f_3 \) prohibits \( f'_2 \), results in that \( f_3 \) prohibits \( f'_2 \) and suggests \( f'_3 \).

When a stakeholder selects two or more features, that decisions can be also propagated. Figure 6 shows the updated FS-Graph after propagating the decision of selecting \( f_1 \) and \( f_3 \). The resulting FS-Graph includes the relationships presented in figures 5.a and 5.c but also other relationships. For instance, notice that feature \( f'_2 \) is suggested by \( f_1 \) but prohibited by \( f_3 \). That means that the feature \( f'_1 \), parent of \( f'_2 \) is non-selectable. Then, the FS-Graph is updated indicating that \( f'_1 \) is prohibited by the selection of \( f_1 \) and \( f_3 \).

Once decisions have been propagated into new relationships in the FS-Graph, we use these relationships to explain why a feature is already selected or is non-
selectable. For instance, in Figure 6 we can identify that \( f_5' \) is non-selectable because the selection of both \( f_1 \) and \( f_3 \), but the feature \( f_6' \) is non-selectable only because the selection of \( f_1 \).

### 3.1.3 Detecting and Explaining Conflict

A configuration conflict occurs when the effect of selecting two or more features in the left-side model invalidates the right-side because some feature become full mandatory and non-selectable.

Using the above example, if only two of the features \( f_1, f_2 \) and \( f_3 \) are selected, the resulting feature model is valid. However, selecting all three features results in conflicting features, i.e. full mandatory features that are non-selectable, e.g. \( f_1' \) is one of them. Figure 7 shows some updated relationships after propagating decisions in FS-Graphs. It includes relationships that show that feature \( f_1' \) is suggested by \( f_2 \) but, at the same time, prohibited by \( f_1 \) and \( f_3 \). Note that, at least, \( f_1', f_2' \) and \( f_3' \) are full mandatory and non-selectable at the same time, i.e. they are conflicting features that invalidate the feature model.

### 3.2 FaMoSA specification

#### 3.2.1 Conflict-tolerant Feature Models

Formal semantics for feature models in FaMoSA are based on semantics proposed by Schobbens et al. [14] but introduces multiple types for each feature and the non-selectable type.
We follow the guidelines of Harel and Rumpe [7] to specify a modeling language $L$ describing the syntactic domain $L_L$, the semantic domain $S_L$ and the semantic function $M_L : L_L \to S_L$, also traditionally written $\llbracket \cdot \rrbracket_L$.

**Definition 1 (Syntactic domain $L_{FM}$)** A feature model $fm \in L_{FM}$ is a 7-tuple $fm = (F, r, DE, w, \lambda, REQ, EXCL)$ such that

- $F$ is the (non empty) set of features
- $r \in F$ is the root
- $DE \subseteq F \times F$ is the decomposition relations which forms a tree. For convenience, we write $f \rightarrow g$ sometimes instead of $(f, g) \in DE$. In addition, we use:
  - $\text{children}(f)$ to denote $\{g | (f, g) \in DE\}$, the set of all direct subfeatures of $f$,
  - $\text{descendants}(f)$ to denote the transitive closure of $DE$, i.e. the set of all direct and indirect subfeatures of $f$,
  - $\text{ancestors}(f)$ to denote $\{g | f \in \text{descendants}(g)\}$
- $w \subseteq F \times FT$ indicates one or more types for each feature, where $FT = \{\text{Optional, Mandatory, NonSelectable}\}$.
- $\lambda \subseteq F \times GT$ indicates the type of feature group for the childrens of each feature. This feature group type can be $GT = \{\text{AlternativeGroup, OrGroup, Aggregation}\}$
- $REQ \subseteq F \times F$ and $EXCL \subseteq F \times F$ are the sets of requires and excludes relations respectively.

Furthermore, each $fm \in L_{FM}$ must satisfy the following well-formedness rules:

- $fm$ is a tree. $r$ is the root and only $r$ has no parent: $\forall f \in F (\exists n' \in F \cdot n' \rightarrow n) \Leftrightarrow n = r$,
- $fm$ is a tree. Therefore all the features $f$ in features are descendants from $r$: $\forall f \in F$ $(f \neq r \Rightarrow \exists n_1, \ldots, n_k \cdot r \rightarrow n_1 \rightarrow \ldots \rightarrow n_k \rightarrow f)$,
- $fm$ is a tree. Therefore $DE$ is acyclic: $\nexists n_1, \ldots, n_k \in \text{features} \cdot n_1 \rightarrow \ldots \rightarrow n_k \rightarrow n_1$
- No feature $f$ has Requires or Excludes constraint to itself. $\forall f \in F ((f, f) \notin REQ \land (f, f) \notin EXCL)$
- All the Alternative and Or feature groups must include more than one feature. $\forall f \in F ((f, \text{AlternativeGroup}) \in \lambda \Rightarrow |\text{children}(f)| \geq 1)$
∀f ∈ F ((f, OrGroup) ∈ λ ⇒ |children (f)| ≥ 1)

**Definition 2 (Semantic domain \( S_{FM} \))** \( S_{FM} \) ≡ \( \mathcal{P}(\mathcal{P}(F)) \)^2, indicating that each syntactically correct model should be interpreted as a set of configurations of products (set of sets of features).

**Definition 3 (Semantic function \( \llbracket fm \rrbracket_{FM} \))** Given \( fm \in \mathcal{L}_{FM} \), \( \llbracket fm \rrbracket_{FM} \) is the set of valid feature configurations \( FC \in \mathcal{P}(\mathcal{P}(F)) \).

**Definition 4 (valid configuration)** A set of features \( c \) is a valid configuration for a feature model \( fm \) if \( c \in \llbracket fm \rrbracket \).

A given configuration is valid regarding (conform to) a feature model \( c \in \llbracket fm \rrbracket_{FM} \) is a configuration \( c \in \mathcal{P}(F) \) such that:

- contain the root: \( \text{root} \in c \),
- justify each feature, that is that if a non-root feature \( g \) is in the configuration, its parent \( f \) must be too:
  \( g \in c \land g \neq r \land g \in \text{children} (f) \Rightarrow f \in c \),
- satisfy the constraints defined by the types of each feature.

\[
\forall f \in c |
(f, \text{NonSelectable}) \notin w \quad \exists g \in \text{children} (f) \bullet (g, \text{Mandatory}) \in w \Rightarrow g \in c
\]

- satisfy the constraints defined by the group type of each feature

\[
\forall f \in c |
(f, \text{AlternativeGroup}) \in \lambda \Rightarrow |\text{children} (f) \cap c| = 1
(f, \text{OrGroup}) \in \lambda \Rightarrow |\text{children} (f) \cap c| \geq 1
\]

- and satisfies the constraints defined by the requires and excludes relationships

\[
\forall f \in c |
(f, g) \in \text{REQ} \Rightarrow g \in c
(f, g) \in \text{EXC} \Rightarrow g \notin c
\]

**Definition 5 (valid feature model)** A feature model \( fm \) is considered valid (not void), if there is at least one valid configuration for it, \( |\llbracket fm \rrbracket_{FM}| > 0 \).
Definition 6 (feature model equivalence) A feature model \(fm_e \in L_{FM}\) is equivalent to another feature model \(fm \in L_{FM}\), if \(\llbracket fm_e \rrbracket_{FM} = \llbracket fm \rrbracket_{FM}\), that is that both models represent the same set of configurations (i.e. their semantics are the same).

Definition 7 (feature model specialization) A given feature model \(fm_s \in L_{FM}\) is a specialization of another feature model \(fm \in L_{FM}\), if \(\llbracket fm_s \rrbracket_{FM} \subseteq \llbracket fm \rrbracket_{FM}\), that is that the first model represents a subset of the configurations defined by the latter.

3.2.2 Feature-solution Graphs (FS-Graphs)

Feature Solution Graphs (FS-Graphs) were proposed by De Bruin et al. [3] to specify the relationships between user-visible features and the corresponding implementation structures and elements. We are extending their ideas using FS-Graphs to specify relationships between features in one level (left-side) to features in next level (right-side). Here we present the syntactic and semantic domain (abstract syntax) of FS-Graphs and in the next section we discuss their semantic function as a feature model specialization.

Definition 8 (Syntactic domain \(L_{FSG}\)) A feature-solution graph \(fsg \in L_{FSG}\) is a 5-tuple \(fsg = (fm, fm', SUG, FOR, PRO)\) such that:

- \(fm \in L_{FM}\) and \(fm' \in L_{FM}\) are left-side and right-side feature models. We write \(F\) to refer to the set of features in \(fm\) and \(F'\) to the set of features in \(fm'\).
- \(FOR \subseteq F \times F'\), \(SUG \subseteq F \times F'\) and \(PRO \subseteq F \times F'\) are the sets of forces, suggests and prohibits relations respectively.

Definition 9 (Semantic domain \(S_{FSG}\)) \(S_{FSG} \triangleq \mathcal{P}(\mathcal{P}(F)) \times \mathcal{P}(\mathcal{P}(F'))\), indicating that each syntactically correct FSG should be interpreted as the relation between configurations of the \(fm\) feature model and a set of configurations of the \(fm'\) feature model.

Definition 10 (Semantic function \(\llbracket fsg \rrbracket_{FSG}\)) Given \(fsg \in L_{FSG}\), \(\llbracket fsg \rrbracket_{FSG}\) returns a set \(\{(c, C) | c \in \llbracket fm \rrbracket_{FM} \land C \subseteq \llbracket fm' \rrbracket_{FM}\}\), such that:

- all the configurations \(c\) are valid configurations of \(fm\) : \(\forall (c, C) \in \llbracket fsg \rrbracket_{FSG} \bullet c \in \llbracket fm \rrbracket_{FM}\).
- all the sets of configurations \(C\) are valid configurations of \(fm'\) : \(\forall (c, C) \in \llbracket fsg \rrbracket_{FSG} \bullet C \subseteq \llbracket fm' \rrbracket_{FM}\).
- each tuple \((c, C)\), satisfy the constraints defined in the forces, suggests and prohibits relations:
  \(\forall (f, f') \in SUG \land f \in c \Rightarrow \forall c' \in C, \text{ if } f' \in \text{children}(g') \land g' \in c' \Rightarrow f' \notin c'\)
∀(f, f') ∈ FOR ∧ f ∈ c ⇒ ∀c' ∈ C, f' ∈ c'
∀(f, f') ∈ PRO ∧ f ∈ c ⇒ ∀c' ∈ C, f' ̸∈ c'

Definition 11 (valid feature solution graph) A given feature solution graph fsg is considered valid (not void), if there is at least one tuple (c, C) ∈ [[fsg]]_{FSG}, such as the corresponding configuration C is not empty: ∃(c, C) ∈ [[fsg]]_{FSG} • C ≠ ∅.

Definition 12 (Feature solution graph equivalence) A feature solution graph fsg_e ∈ L_{FSG} is equivalent to another feature solution graph fsg ∈ L_{FSG}, if [[fsg_e]]_{FSG} = [[fsg]]_{FSG}.

Definition 13 (Feature solution graph specialization) Given fsg_s ∈ L_{FSG} is a specialization of another feature solution graph fsg ∈ L_{FSG}, if [[fsg_s]]_{FSG} ⊆ [[fsg]]_{FSG}. That is that ∀(c, C) ∈ [[fsg_s]]_{FSG} • (c, C) ∈ [[fsg]]_{FSG}.

3.2.3 Propagation of Decisions

As mentioned in section 3.1.2, we propagate the decisions taken by stakeholders in the left-side model by specializing the right-side feature model based on the relationships defined in the FS-Graphs.

However, during specialization, adding a Mandatory or Non-Selectable type to a feature directly referenced by an FS-Graph affects other features too. For instance, if we add a Mandatory type to a feature in an Alternative group, other features in the same group becomes Non-Selectable. These effects may invalidate a feature model when a full mandatory feature becomes de-facto Non-Selectable, and therefore, indicating the existence of a configuration conflict.

The propagation of decisions is a 3 steps process, shown in algorithm 1:

1. Simplify the FS-Graph eliminating the relationships not related to the features included in the configuration of the left-side model, and converting forces relations into sets of suggests.

2. Apply the relations of the FS-Graph marking as Mandatory and Non-Selectable the features on the target side of suggests and prohibits relations, and taking into account sibling features in alternative groups, and child features of non-selectable features.

3. Propagate the effects of adding Mandatory and Non-Selectable types to features by introducing new suggests and prohibits relations into the FS-Graph.

Steps 2 and 3 iterate until a fix point is reached.

Simplification of an FS-Graph First, the FS-Graph is simplified to ease the processing. A simplified FS-Graph fsg' includes only suggests and prohibits relationships involving the features f ∈ c. To consider the forces relationships, a simplified FS-Graph includes the suggests relationships that are equivalent to
Algorithm 1 Propagation of decisions through an FS-Graph

1: procedure propagate(c, fsg)
2:   fsg₀ ← simplify(c, fsg)
3:   i ← 1
4:   while fsgᵢ ≠ fsgᵢ₋₁ do
5:     fsgᵢ = propagateEffects(c, applyFsg(c, fsgᵢ₋₁))
6:     i ← i + 1
7:   end while
8:   return fsgᵢ
9: end procedure

the forces involving the features \( f \in c \). Note that a relationship \( f \xrightarrow{\text{suggests}} f' \) adds a mandatory type to the feature \( f' \) and a \( f \xrightarrow{\text{forces}} f' \) adds a mandatory type to the feature \( f' \) and to all the features \( h \) ancestors of \( f' \). Then, for each forces relationship, it is possible to determine an equivalent set of suggests relationships.

Algorithm 2 Simplification of an FS-Graph

1: procedure simplify(c, fsg)
2:   FOR ← \{ (f, g) \in FOR \mid f \in c \}
3:   SUG ← \{ (f, g) \in SUG \mid f \in c \}
4:   PRO ← \{ (f, g) \in PRO \mid f \in c \}
5:   for all \( f \in c \) do
6:     for all \( (f, f') \in FOR \) do
7:       for all \( g \in \text{ancestors}(f') \) do
8:         addSuggests(f, g)
9:       end for
10:   end for
11: end for
12: return fsg
13: end procedure

Algorithm 2 simplifies an FS-Graph. For brevity, we write \( \text{addSuggests}(f, f') \) to denote \( SUG \leftarrow SUG \cup \{(f, f')\} \), i.e. to add a new \( f \xrightarrow{\text{suggests}} f' \) relationships to an FS-Graph.

Application of effects defined in the FS-Graph After the FS-Graph have been simplified, the specialization process adds Mandatory and Non-Selectable types to the features according to the suggests and prohibits relationships in the FS-Graph \( fsg \) that start from features in the configuration \( c \). In addition, the corresponding suggests and prohibits relationships that represent any modification into the right-side feature model are also added to the FS-Graphs. Note that, after a simplification, all forces relationships were translated to suggests relationships.
relationships.

Feature types and relationships to add to the right-side feature model and FS-Graphs respectively depend on the type of the FS-Graph relation:

For all the \( f \xrightarrow{\text{suggests}} f' \) such that \( f \) is in \( c \):

**effect on** \( f' \): the resulting FM \( fm' \) must include Mandatory as a type of feature \( f \)

**effect on siblings of** \( f' \): if the parent of feature \( f \) in the original FM \( fm \) is an Alternative Group (i.e. a group where just one children can be selected), the resulting FM \( fm' \) must include NonSelectable as a type for all the sibling features \( f' \) of \( f \). In addition, the resulting \( fsg' \) must include a relation that specify that tactic \( t \) prohibits all the sibling features \( f' \) of \( f \).

For all \( f \xrightarrow{\text{prohibits}} f' \) such that \( f \) is in \( c \):

**effect on** \( f' \): the resulting FM \( fm' \) must include NonSelectable as a type of feature \( f \)

**effect on the children of** \( f' \): the resulting FM \( fm' \) must include NonSelectable as a type of every children feature \( f' \) of \( f \). In addition, the resulting FS-Graph \( fsg' \) must include relations indicating that \( t \) prohibits all the children feature \( f' \) of \( f \).

**effect on features that require** \( f' \): the resulting FM \( fm' \) must include NonSelectable as a type of every feature \( f' \) that requires \( f \). In addition, resulting FS-Graph \( fsg' \) must include relations describing that \( t \) prohibits all the features \( f' \) that requires \( f \). In addition, resulting FS-Graph \( fsg' \) must include relations describing that \( t \) prohibits all the features \( f' \) that requires \( f \).

Algorithm 3 describes how the Mandatory and Non-Selectable types are added to the features. We write \( \text{addProhibits}(f, f') \), \( \text{makeMandatory}(f) \) and \( \text{makeNonSelectable}(f) \), to denote respectively that a prohibits relation is added and the addition of mandatory or Non-Selectable to a feature. Also, \( \text{isAlternative}(f) \) and \( \text{thatRequires}(f') \) are used to query whether the group of a feature is an alternative or to obtain the required features by another feature.

**Propagation of Decisions in an FS-Graph** Having propagated the effects of the relations present in the FS-Graph, we now consider the effects that Mandatory or Non-Selective feature types have on other features in the FM.

For all features \( f \) in the source feature model \( fm \) in \( fsg' \):

- \( f \) defines an Alternative group, if \( f \) defines an Alternative feature group, and there are more than one children feature \( f' \) with type Mandatory, the resulting \( fm' \) must include NonSelectable as a type for \( f \). In addition, the
Algorithm 3 Application of an FS-Graph

1: procedure APPLYFSG(c, fsg)
2:   for all \( f \in c \) do
3:     for all \( (f, f') \in SUG \) do
4:       makeMandatory\( (f') \)
5:       if \( \text{isAlternative} \( \text{parent}(f') \) \) then
6:         for all \( g' \in \text{siblings}(f') \) do
7:           makeNonSelectable\( (g') \)
8:           addProhibits\( (f, g') \)
9:       end if
10:   end for
11: end for
12: for all \( (f, f') \in PRO \) do
13:   makeNonSelectable\( (f') \)
14:   for all \( g' \in \text{descendants}(f') \) do
15:     makeNonSelectable\( (g') \)
16:     addProhibits\( (f, g') \)
17:   end for
18: for all \( g' \in \text{thatRequires}(f') \) do
19:   makeNonSelectable\( (g') \)
20:   addProhibits\( (f, g') \)
21: end for
22: end for
23: return fsg
24: end procedure
resulting $fsg'$ must include relations that indicates that all the features $t$ with relations $t$ suggest the children features $t'$ in $fsg$ has prohibits relations to the feature feature $f$ (i.e. they are the cause to make $f$ non-selectable).

$f$ defines an Or or Alternative Group, if $f$ defines an Alternative or Or-Group feature group and all its children feature has non-selectable as a type, the resulting $fm'$ must include NonSelectable as a type for $f$. In addition, the resulting $fsg'$ must include relations that indicates that all the features $t$ with relations $t$ prohibits the children features $f'$ in $fsg$ has prohibits relations to the feature feature $f$ (i.e. they are the cause to make $f$ non-selectable).

$f$ defines an Aggregation Group, if $f$ defines an Aggregation group, $f$ is not a full mandatory feature of $fm$, and there is one or more features $f'$ such that $f'$ has the type Mandatory and NonSelectable, the resulting $fm'$ must include NonSelectable for the feature $f$. In addition, the resulting $fsg'$ must include relations that indicates that all the features $t$ with relations $t$ prohibits the children features $f'$ and the features $t'$ with relations $t'$ suggests the children features $f'$, has prohibits relations to the feature feature $f$ (i.e. they are the cause to make $f$ non-selectable).

$f$ is excluded by a Full Mandatories feature, for all the full mandatory features $f'$ in $fm$, the resulting $fm'$ must include NonSelectable in the types for all the features $f$ excluded by the feature $f'$ in the original FM $fm$. In addition, the resulting FSGraph $fsg$ must include a relation that denotes that the tactics $t$ that suggests the features $f$ are prohibiting $f'$.

$f$ is required by a Full Mandatory feature, for all the full mandatory features $f'$ in $fm$, the resulting $fm'$ must include Mandatory in the types for all the features $f$ required by the feature $f'$ in the original FM $fm$. In addition, the resulting FSGraph $fsg$ must include a relation that denotes that the tactic $t$ suggests the features $f$ required by $f'$.

Algorithm 4 introduces new suggests and prohibits relationships to propagate the effect of adding Mandatory and Non-Selectable types into the target feature model. We write $isMandatory(f)$ and $isNonSelectable(f)$ to query the types of a feature, $suggestorsOf(f')$ and $prohibitorsOf(f')$ to navigate suggests and prohibits relations in the FS-Graphs. Finally, we use

- $fullMandatory(f') \forall g' \in ancestors(f') \bullet (g', Mandatory) \in w$
- $mandatories$ as the set $\{f' \in F'[f', Mandatory) \in w\}$
- $nonselectables$ as the set $\{f' \in F'[f', NonSelectable) \in w\}$
- $conflictingFeatures$ as $\{f' \in F'[fullMandatory(f') \land f' \in nonselectables\}$
Algorithm 4 Propagation of applied features in FS-Graphs

1: procedure PROPAGATE EFFECTS (c, fsg)
2:     for all \( f' \in F' \) do
3:         if isAlternative\( (f') \) then
4:             if |children\( (f') \) \( \cap \) mandatories| \( > 1 \) then
5:                 addProhibits\( (\text{suggestorsOf}(\text{children}(f'), f')) \)
6:             end if
7:         end if
8:         if (isAlternative\( (f') \) \( \lor \) isOr\( (f') \)) then
9:             if children\( (f') \) \( \setminus \) nonselectables = \( \emptyset \) then
10:                addProhibits\( (\text{suggestorsOf}(\text{children}(f'), f')) \)
11:                addProhibits\( (\text{prohibitorsOf}(\text{children}(f'), f')) \)
12:            end if
13:        end if
14:        if isAggregation\( (f') \) \( \land \) \( \neg \) fullMandatory\( (f') \) then
15:            \( g' \leftarrow \text{children}(f') \cap \text{conflictingFeatures} \)
16:            if \( g' \neq \emptyset \) then
17:                addProhibits\( (\text{suggestorsOf}(\text{children}(g'), f')) \)
18:                addProhibits\( (\text{prohibitorsOf}(\text{children}(g'), f')) \)
19:            end if
20:        end if
21:        if fullMandatory\( (f') \) then
22:            for all \( g' \in \text{excludedBy}(f') \) do
23:                addProhibits\( (\text{suggestorsOf}(f'), g') \)
24:            end for
25:            for all \( g' \in \text{requiredBy}(f') \) do
26:                addSuggests\( (\text{suggestorsOf}(f'), g') \)
27:            end for
28:        end if
29:     end for
30: return fsg
31: end procedure
3.2.4 Detecting and Explaining Conflicts

After propagating the decisions of a configuration $c$ into an FS-Graph $fsg$, it is possible to detect if the resulting feature model is invalid, i.e. if the model has features that are full-mandatory and non-selectable. We name here those features conflicting features.

**Definition 14** (conflicting features) In a propagated FS-Graph, the set of conflicting features of the right-side feature model $fm'$ corresponds to the set of features:

$$conflictingFeatures = \{ f' \in F' | fullMandatory(f') \land (f', NonSelectable) \in w \}$$

Updated relationships in the FS-Graph can be used also to detect which features $g$ in a configuration $c$ for the left-side feature model cause the selection or the conflict in a feature $f$ in the right-side feature model.

**Definition 15** (Causes of a feature selection or conflict) In a propagated FS-Graph $fsg$, features CAUSES that cause a conflict or a selection of a feature $f'$ in the right-side feature model is the set of features $f$ of the left-side feature model that have forces or prohibits relationships to $f'$.

$$causes(f') = \{ f | (f, f') \in FOR \lor (f, f') \in PRO \}$$

Note that the resulting FS-Graph, after propagating decisions, allow us to explain which choices of other stakeholders make a feature full mandatory (i.e. selecting it) or non-selectable.
3.3 Formal semantics in Alloy

We have implemented the presented formal semantics using Alloy [9], a modeling language that allow us to analyze and evaluate some properties of the specification using SAT-solvers. Our implementation is based on the proposed by Gheyi [6] to represent the structure of feature models.

3.3.1 Conflict-tolerant Feature Models

As specified in 3.2.1, a feature model $FM$ is composed of a set of features, one of them selected as being the root of the model. Features are linked to each other by relations. Features in an FM are mapped to FeatureTypes (Mandatory, NonSelectabe, or Optional). Additionally, features can group subfeatures in an or, alternative or aggregation relation. Finally, a feature can require or exclude the presence of another feature in valid configurations of the FM.

```alloy
sig FM {
    features : set Name,
    root : one features,
    relations : features lone -> features,
    types : features -> FeatureType,
    groupTypes : features -> one GroupType,
    requires : features -> features,
    excludes : features -> features,
}
sig Name()
abstract sig FeatureType {}
one sig Mandatory, NonSelectable, Optional
    extends FeatureType {}
abstract sig GroupType {}
one sig OrFeatureGroup, Alternative, Aggregation
    extends GroupType {}

We say that a FM is well-formed if all the features except root, are accessible from root across the decomposition relation, and if no features are accessible to themselves by their decomposition relations (do not have self links). In addition, a feature cannot require or include to it self, and no feature can have a feature group Alternative or Or Group when it has less than two children features. Note that a feature model with multiple types (e.g. mandatory and non-selectable) is well-formed.

pred wellFormedFM( fm : FM ) {
    all f:fm.features - fm.root
    | f in (fm.root)."(fm.relations)
    no f:fm.features
    | f in f."(fm.relations)
    no f:fm.features | f in fm.requires[f]
    no f:fm.features | f in fm.excludes[f]
    no f:fm.features
    | fm.groupTypes[f] in Alternative OrFeatureGroup
    and #fm.relations[f]<1
}
```
Feature configurations

In our model, features with conflicting feature-types (e.g., a feature being both mandatory and non-selectable) can be well-formed. However, not all well-formed feature models can have valid configurations. We define a configuration as a set of feature (names).

\[
\text{sig Configuration} \{ \\
\quad \text{value : set Name} \\
\}\n\]

A configuration \( c \) is valid with respect to a feature model \( fm \), if the configuration includes a set of features that satisfies the constraints of \( fm \). The validity of a configuration is decomposed into whether the configuration satisfies the implicit and explicit constraints, and whether it satisfies the relations, group types and feature types of the model.

\[
\text{pred isValid} \ ( fm : \text{FM}, \ c : \text{Configuration} ) \{ \\
\quad \text{satisfyImplicitConstraints} \ [ fm, \ c ] \\
\quad \text{and satisfyExplicitConstraints} \ [ fm, \ c ] \\
\quad \text{and satisfyRelations} \ [ fm, \ c ] \\
\quad \text{and satisfyGroupTypes} \ [ fm, \ c ] \\
\quad \text{and satisfyFeatureTypes} \ [ fm, \ c ] \\
\}\n\]

Implicit constraint satisfaction checks that the configuration contains only features present in the feature model, and that it includes the root feature.

\[
\text{pred satisfyImplicitConstraints} \ ( fm : \text{FM}, \ c : \text{Configuration} ) \{ \\
\quad fm.\text{root in c.value} \\
\quad c.\text{value in fm.features} \\
\}\n\]

Explicit constraint satisfaction, on the other hand, checks that the requires and excludes constraints defined in the feature model are respected.

\[
\text{pred satisfyExplicitConstraints} \ ( fm : \text{FM}, \ c : \text{Configuration} ) \{ \\
\quad \text{all f : c.value} \{ \\
\quad\quad fm.\text{requires}[ f ] \text{ in c.value} \\
\quad\quad fm.\text{excludes}[ f ] \neq \text{none} \\
\quad\quad \Rightarrow ( fm.\text{excludes}[ f ] \notin \text{c.value} ) \\
\quad\} \\
\}\n\]

A feature configuration \( c \) satisfies the relation-based constraints in a FM \( fm \) when none feature \( f \) is included in \( c \) and its parent is not included in \( c \). That is, if a feature is included in the configuration, the parent feature in the FM must be included as well.

\[
\text{pred satisfyRelations} \ ( fm : \text{FM}, \ c : \text{Configuration} ) \{ \\
\quad \text{no f : c.value} \{ \\
\quad\quad fm.\text{relations}.(f) \text{ in c.value} \\
\quad\} \\
\}\n\]

The predicate \text{satisfyGroupTypes} checks that the configuration respects the semantics of the OrGroup and AlternativeGroup types. A feature configuration \( c \) satisfies the alternative group types in a FM \( fm \) when, for all the features \( f \in c \), if the group type of \( f \) is Alternative then one of its children is also in \( c \). In addition, a configuration \( c \) satisfies the Or group types in a FM \( fm \) when, for all the features \( f \) in \( c \), if the group type of \( f \) is Or then at least one
of its children is also in \( c \). The Aggregation group type does not impose any constraint to the configurations.

\[
\text{pred satisfyGroupTypes( fm : FM, c : Configuration ) } \{
    \text{no f : c.value \{ }
    \text{fm.groupTypes[f] = Alternative}
    \text{and #(fm.relations[f] & c.value) \neq 1 }
    \text{\}}
    \text{no f : c.value \{
    fm.groupTypes[f] = OrFeatureGroup
    #(fm.relations[f] & c.value) = 0
    \}}
\}
\]

NonSelectable and Mandatory feature types also defines constraints to valid configurations. A feature configuration \( c \) satisfies the NonSelectable type in a FM \( fm \) when, for all the features \( f \) in \( c \), none of them is defined as NonSelectable in \( fm \). In addition, a configuration \( c \) satisfies the Mandatory type in a FM \( fm \) when, for all the features \( f \) in \( c \), all the children features \( f' \) of the feature \( f \) defined with type mandatory in \( fm \) are also included in \( c \). The Optional feature type does not impose constraints to features in a valid configuration.

\[
\text{pred satisfyFeatureTypes( fm : FM, c : Configuration ) } \{
    \text{no f : c.value \{ }
    \text{| NonSelectable in fm.types[ f ] }
    \text{no disj f : c.value, f' : fm.features \{ }
    \text{f' in fm.relations[ f ]}
    \text{Mandatory in fm.types[ f ]}
    \text{f' \notin c.value }
    \text{\}}
\}
\]

Notice that relations, feature types and feature group types in a feature model constraint the space of valid configurations regarding that model. Semantics for a FM \( fm \) correspond to the set of configurations that satisfy all the implicit and explicit constraints defined by its relations, feature types and group types, i.e. they correspond to the set of configurations that are valid to it.

\[
\text{fun semantics( fm : FM ) : set Configuration } \{
    \{ c : Configuration | isValid [ fm , c ] \}
\}
\]

A dead feature is a feature that is not included in any valid configuration, whereas a full mandatory feature is one that is present in all valid configurations.

\[
\text{fun deadFeatures( fm: FM ) : set Name } \{
    \{ f: Name |}
    \text{all c : semantics[ fm ] |}
    \text{f \notin c }
    \}
\}
\]

\[
\text{fun fullMandatory( fm: FM ) : set Name } \{
    \text{fm.root ."^( fm.relations :> fm.types.( Mandatory ) ) ”}
    \}
\]

In our semantics, a non-selectable feature is a dead feature. In addition, any feature that is excluded by a full mandatory feature (i.e. excluded by a feature that must be included in all valid configurations) is also a dead feature.
A valid feature model is a feature model that has valid configurations (i.e. a non-empty semantic set). Invalid feature models are those where it is not possible to define a configuration that is valid regarding them (i.e. an empty semantic set).

A feature model including full-mandatory features typed as non-selectable is an invalid feature model because it is not possible to create a valid configuration including a feature, and without including the same feature. The following conflictingFeatures function returns the set of full-mandatory features that have a NonSelecteable as one of their types (i.e. a set of features that make a feature model invalid).

fun conflictingFeatures : set Name {
    ( f : fullMandatories | NonSelecteable in fm.types[f] )
}

3.3.2 Feature-solution Graphs (FS-Graphs)

Having defined the conflict-tolerant feature models, we now show how we specify feature solution graphs in FaMoSA: A Feature-Solution Graph (FS-Graph) comprises a source FM, a target FM and sets of relations between features in the source FM to features in the target FM. In our specification, the source is an FM representing architectural tactics and the target an FM representing design alternatives. Each type of relation (forces, suggests and prohibits) is represented by its own mapping.

sig FSGraph {
    source : FM,
    target : FM,
    forces : Name -> Name,
    suggests : Name -> Name,
    prohibits : Name -> Name,
}

A FS-Graph is well formed if the source and target feature models are well formed, and if all the relations (i.e. forces, suggests, and prohibits) have domain in the source features and range in the target features.

pred wellFormedFSGraph : FSGraph {
    wellFormedFM [ fsg.source ]
    wellFormedFM [ fsg.target ]
    no f : Name, f' : fsg.forces[f] | f \notin fsg.source.features and f' \notin fsg.target.features
    no f : Name, f' : fsg.suggests[f] | f \notin fsg.source.features and f' \notin fsg.target.features
    no f : Name, f' : fsg.prohibits[f] | f \notin fsg.source.features and f' \notin fsg.target.features
}
In FaMoSA, semantics of an FS-Graphs are an specialization of the right-side feature model. Given a FS-Graph \( fsg \) and a configuration \( c \) that is valid regarding the source FM in \( fsg \), the relations in \( fsg \) represent the following modifications:

- **forces relations.** A forces relation \( f \xrightarrow{\text{forces}} f' \) in \( fsg \) denotes that the types of features in \( fsg \) must be in \( f' \) in the target FM of \( fsg \) must be converted to full-mandatory when \( f \) is included in the configuration \( c \).

- **suggests relations.** A suggests relation \( f \xrightarrow{\text{suggests}} f' \) in \( fsg \) indicates that the feature \( f' \) in the target FM of \( fsg \) must be typed as mandatory just in the feature group where it belongs when \( f \) is included in \( c \).

- **prohibits relations.** A prohibits relation \( f \xrightarrow{\text{prohibits}} f' \) denotes that the feature \( f' \) in the target FM of \( fsg \) must be typed as a non selectable feature when \( f \) is included in \( c \).

We define a predicate `appliedFSGraphToFM` representing the specialization of an FS-Graph's target feature model. The predicate relates a valid configuration \( c \) of the source FM of a FS-Graph \( fsg \), and the target of the FS-Graph before \( (fm) \) and after \( (fm') \) the specialization. The specialization defined in FaMoSA, are base on changing the feature types only, and this only adding feature types to \( fm' \) as directed by the relations present in \( fsg \): features will now include Mandatory if exists a relation \( f \xrightarrow{\text{forces}} f' \) in \( fsg \) and \( f \) is in \( c \); if exists a relation \( f \xrightarrow{\text{forces}} g \) in \( fsg \), the feature \( f \) is in \( c \) and the feature \( f' \) is an ancestor of \( g \); or if exists a relation \( f \xrightarrow{\text{suggests}} f' \) and the feature \( f \) is in \( c \). They will now include Non-Selectable if exists a relation \( \rightarrow_{\text{prohibits}} \) \( f' \) in \( fsg \) and \( f \) is included in \( c \).

```plaintext
pred appliedFSGraphToFM
(c: Configuration, fsg: FSGraph, fm, fm': FM) {
    fm'.root = fm.root
    fm'.features = fm.features
    fm'.relations = fm.relations
    fm'.types = fm.types
    + { f : fsg.forces[ c.value ],
        type : Mandatory }
    + { f : fsg.target.features,
        type : Mandatory
        | some g : Name
        | g in fsg.forces[ c.value ]
        and (f in ancestors[ fsg.target, g])
    }
    + { f : fsg.suggests[ c.value ],
        type : Mandatory }
    + { f : fsg.prohibits[ c.value ],
        type : NonSelectable }
    fm'.groupTypes = fm.groupTypes
    fm'.requires = fm.requires
    fm'.excludes = fm.excludes
}
```
fun ancestors ( fm : FM , f : Name ) : set Name {
  {f' : fm. features | f in f'.^( fm. relations) }
}

During the configuration process, when the decisions are propagated, the semantics of the right-side feature model must be preserved. We use this appliedFSGraphToFM predicate in Alloy to evaluate that the following predicates and transformations, that we use to propagate the decisions, preserve the semantics of the right-side model.

3.3.3 Propagation of decisions

Propagation of decisions, as explained in the section 3.1.2, allows FaMoSA to detect conflicting features in a configuration process. Finding out which are the choices in the left-side FM that cause a conflict is a simple matter of tracing back the relations defined in the FS-Graph from the conflicting features back to their corresponding features in the left-side.

As mentioned, the steps of the propagation are as follows, from a valid configuration of the source model in the FS-Graph:

1. Simplify the FS-Graph by converting forces relations into sets of suggests;
2. propagate the relations of the FS-Graph by applying it, marking as Mandatory and NonSelectable the features on the target side of suggests and prohibits relations, and taking into account sibling features in alternative groups, and child features of non-selectable features;
3. propagate the effects of the types of features on their children, and the effect of explicit relations (requires and excludes) on Full Mandatory features.

Steps 2 and 3 are iterated until a fix point is reached. In what follows, we detail each of the steps.

Simplification of an FS-Graph A Simplified FS-Graph \( fsg' \) of a FS-Graph \( fsg \) and configuration \( c \) includes only the relations related to features \( f \) in \( c \) and the a set of suggests relations equivalent to the forces in \( fsg \). That means that source FM, target FM in \( fsg' \) are the same in \( fsg \). In addition, forces and prohibits relations in \( fsg' \) are only those relations in \( fsg \) that has domain in \( c.value \). Finally, suggests relations in \( fsg \) include the suggests relations in \( fsg \) with domain in \( c.value \) plus new suggest relations to the features \( f \) forced by the features \( x \) in \( c \), and to the their ancestor features \( g \).

pred convertForcesIntoSuggests
( c : Configuration , fsg , fsg' : FSGraph ) {
  fsg'.source = fsg.source
  fsg'.target = fsg.target
  fsg'.forces = c.value <: fsg.forces
  fsg'.suggests = c.value <: fsg.suggests
}

26
Application of effects defined in an FS-Graph  A *relation-propagated FS-Graph* is a FS-Graph whose target feature model and relations represent the effects of adding a Mandatory or NonSelectable type to a feature on other features.

The predicate `appliedEffectsIntoFM` represents the transformation of `fm` into `fm'` after propagating the effects of the relations of a given FS-Graph `fsg` and a configuration `c`.

```plaintext
def appliedEffectsIntoFM( c : Configuration, fsg : FSGraph, fm, fm' : FM ) {
    all t : c.value, f : fsg.suggests[ t ] {
        Mandatory in fm'.types[ f ]
        fm.groupTypes[ parent[ fm, f ] ] = Alternative
        => all f' : siblings[ fm, f ] |
            NonSelectable in fm'.types[ f' ]
    }
    all t : c.value, f : fsg.prohibits[ t ] {
        NonSelectable in fm'.types[ f ]
        all f' : children[ fm, f ] |
            NonSelectable in fm'.types[ f' ]
        all f' : thatRequires[ fm, f ] |
            NonSelectable in fm'.types[ f' ]
    }
}
```

Also, we define a `appliedEffectsIntoFSG` predicate that describe the FS-Graph `fsg'` that results after propagating the effects of the relations in `fsg` using a configuration `c`.

```plaintext
def appliedEffectsIntoFSG( c : Configuration, fsg, fsg' : FSGraph ) {
    let fm = fsg.target {
        all t : c.value, f : fsg.suggests[ t ] {
            fm.groupTypes[ parent[ fm, f ] ] = Alternative
            => { all f' : siblings[ fm, f ] |
                t -> f' in fsg'.prohibits }
        }
        all t : c.value, f : fsg.prohibits[ t ] {
            all f' : children[ fm, f ] |
            t -> f' in fsg'.prohibits
            all f' : thatRequires[ fm, f ] |
            t -> f' in fsg'.prohibits
        }
    }
}
```
Propagation of decisions in an FS-Graph  Having propagated the effects of the relations present in the FS-Graph, we now consider the effects that Mandatory or Non-Selectable feature types have on other features in the FM.

We define the predicate \( \text{propagatedDecisionsIntoFM} \) representing the transformation of \( fm \) into \( fm' \) after propagating the effects of modifications resulting of applying the relations of a given FS-Graph \( fsg \) and a configuration \( c \).

\[
\text{pred propagatedDecisionsIntoFM} ( c : \text{Configuration}, \ fsg : \text{FSGraph}, \ fm, \ fm' : \text{FM} ) \{
\begin{align*}
& \text{all } f : \text{fm.features} \\
& \quad | \text{fm.groupTypes(f) = Alternative} \\
& \quad \text{and } \#(\text{children( fm, f )} \\
& \quad \text{& fm.types.(Mandatory)}) > 1 \\
& \quad \Rightarrow \ \text{NonSelectable in fm'.types[ f ]}
\end{align*}
\]

\[
\begin{align*}
& \text{all } f : \text{fm.features} \\
& \quad | \text{fm.groupTypes(f) in (Alternative + OrFeatureGroup)} \\
& \quad \text{and } \#(\text{children( fm, f )} \\
& \quad \text{& fm.types.(NonSelectable) }) = \#\text{children( fm, f )} \\
& \quad \Rightarrow \ \text{NonSelectable in fm'.types[ f ]}
\end{align*}
\]

\[
\begin{align*}
& \text{all } f : \text{fullMandatories( fm )} \\
& \quad \text{all } f' : \text{excluded( fm, f )} \\
& \quad | \ \text{NonSelectable in fm'.types[ f' ]}
\end{align*}
\]

\[
\begin{align*}
& \text{all } f : \text{fullMandatories( fm )} \\
& \quad \text{all } f' : \text{fm.requires(f)} \\
& \quad | \ f' \ in \text{fullMandatories( fm' )}
\end{align*}
\]

\[
\begin{align*}
& \text{all } f : \text{fm.features} \\
& \quad | \text{fm.groupTypes(f) in (Aggregation)} \\
& \quad \text{and } \#\text{children( fm, f )} >= 1 \\
& \quad \text{and } \#(\text{children( fm, f )} \text{& fm.types.(Mandatory) & fm.types.(NonSelectable)}) = \#\text{children( fm, f )} \\
& \quad \Rightarrow \ \text{NonSelectable in fm'.types[ f ]}
\end{align*}
\]

Also, we define a predicate \( \text{propagatedDecisionsIntoFSGraph} \) representing the transformation of \( fsg \) into \( fsg' \) after propagating the effects resulting of applying the relations of \( fsg \) and a configuration \( c \).

\[
\text{pred propagatedDecisionsIntoFSGraph} ( c : \text{Configuration}, \ fsg, \ fsg' : \text{FSGraph} ) \{
\begin{align*}
& \text{let fm = fsg.target} \\
& \quad \text{all } f : \text{fm.features} \\
& \quad | \text{fm.groupTypes(f) = Alternative} \\
& \quad \text{and } \#(\text{children( fm, f )} \\
& \quad \text{& fm.types.(Mandatory)}) > 1 \\
& \quad \Rightarrow \{ \\
& \quad \quad \text{all } t : \text{fsg.suggests.(children( fm, f )} \\
& \quad \quad \text{& fm.types.(Mandatory))} \\
& \quad \quad | \ t -> f \ \text{in fsg'.prohibits}
\end{align*}
\]

\[
\begin{align*}
& \text{all } f : \text{fm.features}
\end{align*}
\]
3.3.4 Detecting and Explaining conflicts

After all the modifications of applying a FS-Graph have been propagated, conflicts caused by during the configuration process can be detected determining full-mandatory features typed as non-selectable.

```plaintext
fun conflicts ( fsg : FSGraph ) : set Name {
    conflictingFeatures[ fsg.target ]
}
```

In addition, the updated relations in the FS-Graph can be used to determine which features in the left-side feature model cause that a feature in the right-side model become mandatory or non-selectable.

```plaintext
fun causes( fsg : FSGraph , f : Name ) : Name {
    { g : fsg.source.features |
        f in fsg.suggests[ g ]
        or f in fsg.prevents[ g ]
    }
}
```

4 Conclusion

FaMoSA is our multi-level configuration process to detect conflicts and causes using feature-solution graphs and feature models that tolerate conflicts. It is based on feature models that can be specialized without removing features, because each feature can be typed at the same time as mandatory, optional and non-selectable. We use FS-Graph to specify how features in one level affect features in other levels. FaMoSA’s FS-Graphs support forces, suggests and prohibits relationships, our configuration process uses these relationships to propagate decisions from one level (left-side feature model) to the next one (right-side feature model).
In terms of future work, FaMoSA does not support disjunctive relationships (e.g. $f$ forces $f'$ or $g'$) nor arbitrary inclusions of CNF constraints on its feature models. Extended feature models, such models with cardinalities [5], are not supported neither. We are currently working on using FS-Graphs to maintain traceability during specialization of feature models with these extensions.

In addition, multi-level configuration processes may involve more than two feature models in sequence or in combinations of sequential and parallel steps. Then, solving a conflict requires to determine causes and effects in each step and propose alternative configurations to each stakeholder. We consider that type of processes can be supported using diverse topologies of FS-Graphs. Applying feature-solution graphs to detect conflicts and propose fixes in these scenarios remains the subject of future work.

Acknowledgements

Jaime Chavarriaga is a recipient of a COLCIENCIAS (Colombian Department of Science, Technology and Innovation) fellowship. Carlos Noguera is funded by the FWO-AIRCO project of the Fonds Wetenschappelijk Onderzoek of the Flemish Region.

References


