Reasoning about Feature Models in Higher-Order Logic

Mikoláš Janota and Joseph Kiniry
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
mikolas.janota@ucd.ie, joseph.kiniry@ucd.ie

Abstract

A mechanically formalized feature modeling meta-model *is presented. This theory is a generic higher-order formalization of a mathematical model synthesizing several feature modeling approaches found in the literature. This meta-model support not only a better understanding of the various approaches to feature modeling, but also supports reasoning* about *and* within *feature model approaches, feature models, and on feature trees and their configurations.*

1 Introduction

While recently performing a survey of the use of formal methods in software product line (SPL) research [CN02, PBv05] we found that there were several different approaches to feature modeling. While most of these approaches stem from FODA [KCH+90], some have very subtle differences (either intentionally or accidentally) and, frequently, precise explanations of their semantics are unavailable [FFB02].

Because we wanted to deeply understand feature modeling, we decided to mechanically formalize a *feature modeling meta-model*, a generic higher-order formalization of a mathematical model synthesizing several feature modeling approaches found in the literature. This meta-model has not only helped us better understand the various approaches to feature modeling, but also reason *about* and *within* feature models.

Additionally, as part of the Mobius project, we are leading the development of the Mobius Program Verification Environment (PVE), an Eclipse-based platform for designing, testing, performing various kinds of static analyses, and automatically and interactively formally verifying Java program and bytecode. As we have already integrated PVS into this environment, using an executable theory from within Eclipse is a very natural. Using the underlying formalism of feature meta-models described in this paper, we have an opportunity to make the Mobius PVS a feature-aware programming environment, and perhaps even integrate feature-oriented programming and the main programming paradigm of Mobius, that of proof-carrying code. To bridge these two worlds we plan to assign features to Java components in a manner similar to Batory's AHEAD tool suite on a feature-aware component model.

Thus this work contains several contributions. First, we have mechanically formalized in a higher-order logic (HOL) a "feature model meta-model" that integrates properties found in several feature modeling approaches from the literature. To provide evidence that our meta-model is sufficiently rich to reason about, and with, existing feature models, we have formalized specific existing feature model frameworks in our meta-model. Complementing this work, we have formulated and proven a number of interesting meta-theoretical lemmas about different feature modeling approaches, which helps one understand how different approaches relate to one other. To compare feature models, we have identified, formulated and proven some interesting feature model transformations. Additionally, because we are reasoning with/in HOL, we can express complex dependencies about and within a feature model, and can reason about an unbounded number of configurations. Finally, because we have mechanically realized the theory in the higher-order prover PVS, we have an executable model that can be used in conjunction with standard software engineering tools like integrated development environments.

Before describing our formalism, we first review in Section 2 the relevant background material on feature modeling and, in particular, formalizing about, and within, feature models.

The meta-model is then described in detail in Section 3 and, using that meta-model, we formalize specialization as a model relation.

To provide evidence that our meta-model is flexible and has utility, in Section 4, we realize a number of feature model examples from the literature. In particular, Czarnecki et al.'s cardinality and cloning-based scheme(s) (with attributes and model transformations) are formalized. Next, concrete feature models and configurations are described and checked within our theory.

In Section 6 we review the existing feature modeling approaches from which we derive, and against which we compare, our formal meta-model.

Finally, we reflect on this work and propose some next steps in Section 5. In particular, in the interest of full disclosure, we highlight this work's limitations and strengths in plain language.

2 Feature Modeling

We presume the reader is generally aware of the existing work on feature models/modeling, particularly its origins in FODA [KCH+90] and the general ideas of *domain engineering* and *application engineering*. We only highlight the specific work that informs this research, providing a foundation for our meta-model.

The technical report introducing the *feature oriented domain analysis* [KCH+90] represents the seminal work in feature modeling used in product lines. Among other things, the report has introduced the following basic concepts.

Feature diagram: A tree of features, a graphical representation of a feature hierarchy. The parent-child relation has the meaning that the child cannot be selected into a configuration unless the parent is also included. Features are categorized as being either *mandatory*, *optional*, or *alternative*.

Each *alternative* feature is part of a group of alternative features under a certain parent, meaning that exactly one of these children has to be selected whenever this parent is selected. An *optional* feature may or not may be included into the configuration when its parent is selected. A *mandatory* feature has to be selected whenever its parent is selected.

Composition rules: This mechanism enables the expression of mutual dependency (*requires*) and mutual exclusion (*mutex-with*) relations between two arbitrary features. Using these relations one expresses relationships between features across the tree structure.

Czarnecki et al. provides an extension of FODA, extending it with groups of *or* sub-features[1] [CE00].

This work has further evolved in the *cardinality-based notation* [CHE04b, CHE04a], based on a case study that we use later [CBUE02]. In this approach, constraints on features are expressed using admissible cardinalities, as in, for example, Entity-Relation (ER) or UML modeling.

[1]At least one of the features from an *or* sub-group has to be selected whenever their parent is selected.

Each feature is either a member of a feature group, what is called a *group feature*, or it not, and is known as a *solitary feature*. Each feature group has a cardinality specifying how many features out of that group can be selected when the parent of that group is selected. A set of admissible cardinalities is expressed as list of intervals of natural numbers. Each *solitary feature* has cardinality associated with it specifying how many copies of that particular feature can be selected whenever its parent is selected. Selecting multiple copies of the same feature into a configuration is called *feature cloning*.

For example, a solitary feature with the cardinality $[1..1]$ corresponds to the mandatory classification; a group with the cardinality $[1..1]$ corresponds to an alternative group.

A feature model can be broken up into multiple trees (a forest), where a root of a tree can be referenced from the inside of another tree as a sub-feature. Redundancies are thus avoided, as a single tree can be referenced multiple times. The use of a forest also provides a modularization mechanism for large feature models.

Also, *attributes* (also called *properties*) of features are used to represent variabilities with a large domain, such as numeric values or strings.

In another article describe a number of *specializations steps*, i.e., operations on the diagram that lead to a more specialized diagram [CHE04b]. Examples of specialization steps include eliminating admissible cardinalities of a certain feature, or assigning a value to an attribute.

3 Formalization

For the purpose of formalization we must decide how to model the individual concepts of all of the feature modeling approaches summarized in Section 2. In the models that we have examined in the literature, a feature is simply an entity with possibly some properties, such as "name". This led us to formalizing a feature as a record with a set of attributes, where each attribute models a property of that particular feature.

Utilizing this definition, we define a feature configuration as the set of features that are selected and the values of their attributes. Subsequently, a feature model is defined as a function that determines the set of valid configurations. In other words, we regard a feature model as an "oracle" that responds either valid or invalid for given a configuration.

What follows defines a feature meta-model and a feature model. We first formalize a type system of features, as the PVS theorem prover provides a typed higher-order logic.

Definition 3.1. *First we introduce the following types that characterize the entities of our meta-model* $\mathbb{M}$*. Let* $\mathcal{F}$ *be* *the (uninterpreted) type of features, $\mathcal{I}$ the type of attribute identifiers, $\mathcal{AT}$ the type of attribute types, and $\mathcal{AV}$ the type of attribute values.*

An value typing function *assigns a type to every attribute value:* $\mathtt{typing} : \mathcal{AV} \rightarrow \mathcal{AT}$

*Let $\mathcal{T}$ be a set of feature types, where each type is a pair composed of a set of identifiers, representing the attributes, and a function assigning types to these identifiers (*an attribute typing function*). We express this as the following dependent type:*

$$\mathcal{T} \equiv (ids : \mathcal{P}(\mathcal{I})) \times (att : ids \rightarrow \mathcal{AT})$$

In the following, for a type $t \in \mathcal{T}$ we will use the notation $t.ids$ and $t.att$ to access the identifiers in t and the attribute typing function of t, respectively.

A feature typing function *assigns types to all features:* $\mathtt{type} : \mathcal{F} \rightarrow \mathcal{T}$

A value assignment *is a function assigning values to the attributes of all the features in the domain. The type of a value assignment function aa is a subtype of the function type*

$$\mathcal{F} \rightarrow (D \rightarrow \mathcal{AV}) \text{ where } D \subseteq \mathcal{I}$$

such that, for any feature f, $dom(aa(f)) = \mathtt{type}(f).ids$ and, for all $id \in dom(f)$, $\mathtt{typing}(aa(f)(id)) = \mathtt{type}(f)(id)$.

In plain language, this means that the value assignment adheres to the typing prescribed by the functions `typing` *and* `type`*. We will refer to this type, a type of all value assignments, as $\mathbb{A}$.*

The features selected in a configuration are modeled by a selection function *which is of the type* **select***, defined as a function from features to booleans:* $\mathbf{select} \equiv \mathcal{F} \rightarrow \mathbb{B}$

Finally, a restriction functions *is at the heart of every feature model. Given a configuration (the values of attributes and given a set of selected features), the restriction function indicates whether this configuration is admissible or not. This is expressed as the following type*

$$\mathbf{restr} \equiv \mathbf{select} \times \mathbb{A} \rightarrow \mathbb{B}$$

Now that we have defined all the necessary types, we define the meta-model itself.

Definition 3.2. *Given Definition 3.1, a* feature meta-model *$\mathbb{M}$ is a set of restriction functions and a* feature model *is a particular selected restriction function $\mathcal{M} \in \mathbb{M}$.*

Example 3.3. *This example illustrates how the functions defined above can be used. We wish to model the record* `f1 = [ name : String ]` *and the assignment* `f1.name = ''Hi''`*.*

The following formula states that the value "Hi" is of the type string.

$$\mathtt{typing}(\text{“Hi”}) = \mathtt{string}$$

Then it is possible to express that the type of feature f_1 is a record that contains a single attribute identified by the identifier `name`*, whose type is string.*

$$\mathtt{type}(f_1) = \langle\{\mathtt{name}\}, \lambda id : \{\mathtt{name}\} \bullet \mathtt{string}\rangle$$

The following formula states that the value assignment function aa has assigned the value `''Hi''` *to the attribute* `name` *of the feature f_1.*

$$aa(f_1)(\mathtt{name}) = \text{“Hi”}$$

Example 3.4. *Many feature modeling approaches, such as the original FODA model, do not enable expressing any restrictions on attributes. This limitation is formalized by the following formula:*

$$\forall s : \mathbf{select}; a_1, a_2 : \mathbb{A} \bullet restr(s, a_1) = restr(s, a_2)$$

Czarnecki et al. introduced the notion of staged configuration, an approach to configuration where the model is gradually specialized until a model enabling a single configuration is left [CHE04b] . Using the above formalization, it is straight-forward to formally define specialization.

Definition 3.5. *Let m_1 and m_2 be feature models defined on the same set of attributes and feature domain. Let $restr_1$ and $restr_2$ be their restriction functions respectively. Then m_2 is a* specialization *of m_1 if and only if all the admissible configurations by $restr_2$ are also admissible by $restr_1$. This is realized as the following predicate on restriction functions.*

$$\begin{aligned}&specialization?(restr_2, restr_1 : \mathbf{restr}) \equiv\\&\quad \forall s : \mathbf{select}; a : \mathbb{A} \bullet restr_2(s, a) \Rightarrow restr_1(s, a)\end{aligned}$$

This definition is more liberal than that presented by Czarnecki et al. as a specialization that does not reduce the set of possible configuration is still considered a specialization here.

4 Applying the Theory

In the previous section we have formally defined the concept of the feature model. This section brings feature diagrams into this context. More specifically, we address the translation of a feature diagram, (a labeled graph on features), to a feature model, which is a restriction function on configurations (see Definitions 3.1 and 3.2).

This translation is accomplished in two steps. First we formalize a specific type of feature diagram as a mathematical construct. Second, we define a function that takes this construct and returns a restriction function corresponding to the semantics of that diagram.

Furthermore, we illustrate on several lemmas and examples how we can reason and work with this formalization.

Please note that we do not provide the full account of the proofs here due to space limitations. Rather sketches of the proofs written in the proof assistant PVS are provided. As in the previous section, we utilize a standard mathematical notation to describe the PVS formalization. We encourage the reader to download our PVS theories and proofs from our research group's homepage for more details.

4.1 Feature Diagram Formalization

In this section we formalize the feature model presented by Czarnecki et al. using our meta-model $\mathbb{M}$, as mechanically realized in the PVS theorem prover [CHE04b].

From the meta-modeling point of view, this model represents an interesting challenge because we must model the ability to clone a feature. We have chosen to model this notion by introducing the concept of *clone groups*. A clone group contains all the possible clones of the feature that we wish to be cloneable. In other words, such a group represents a pool of clones for that particular feature.

The only restriction that we impose on a clone group is that all of its members must be of the same type. Hence, for each solitary feature in the original model, we introduce a clone group whose members (the clones) are of the type of that feature. Since the grouped features in the original model are aggregated into a group, it is only natural to make them members of groups in our model as well. We refer to these kind of groups simply as *feature groups*.

Admissible cardinalities must also be formalized. They are realized by labeling each group with the set of its admissible cardinalities. In the original work cardinalities are represented as lists of intervals of natural numbers; in the model presented here, however, we enable any subset of natural numbers. While there is a difference between these two representations, it is not crucial for this work.

Definition 4.1. *Given the types defined in Definition 3.1, and given an additional type $\mathcal{G}$, the type of groups, a feature tree is a tuple*

$$\mathbf{TREE} \equiv \langle R, \mathcal{L}_T, \mathcal{L}_C, \text{groups}, \text{members}\rangle$$

where $R : \mathcal{F}$ is the root concept of the feature tree, $\mathcal{G}$ is a set of groups, where each group is labeled by $\mathcal{L}_T$ according to its type, and by $\mathcal{L}_C$ determining its admissible cardinalities,

$$\begin{aligned}\mathcal{L}_T &: \mathcal{G} \to \mathit{feature} \mid \mathit{clone}\\ \mathcal{L}_C &: \mathcal{G} \to \mathcal{P}(\mathbb{N})\end{aligned}$$

The structure of the diagram is defined by the functions groups *and* members*. For each feature f,* groups *determines which groups f owns and the function* members *determines the features that belong to a given group.*

$$\begin{aligned}\text{groups} &: \mathcal{F} \to \mathcal{P}(\mathcal{G})\\ \text{members} &: \mathcal{G} \to \mathcal{P}(\mathcal{F})\end{aligned}$$

These two functions have to satisfy the following conditions: (1) the root concept does not belong to any group, (2) no feature belongs to more than one group, (3) each group is used for partitioning at most one feature and, finally, (4) members of each clone group must have the same type.

$$\forall g : \mathcal{G} \bullet R \notin \text{members}(g) \tag{1}$$

$$\begin{aligned}&\forall g_1, g_2 : \mathcal{G} \bullet g_1 \neq g_2 \Rightarrow\\ &\quad \text{members}(g_1) \cap \text{members}(g_2) = \emptyset\end{aligned} \tag{2}$$

$$\begin{aligned}&\forall f_1, f_2 : \mathcal{F} \bullet f_1 \neq f_2 \Rightarrow\\ &\quad \text{members}(f_1) \cap \text{members}(f_2) = \emptyset\end{aligned} \tag{3}$$

$$\begin{aligned}&\forall g : \mathcal{G} \bullet \mathcal{L}_T(g) = \mathit{clone} \Rightarrow\\ &\quad \forall f_1, f_2 \in \text{members}(g) \bullet \mathtt{type}(f_1) = \mathtt{type}(f_2)\end{aligned} \tag{4}$$

Now that we have defined what a feature diagram is, we can define the standard parent-child relationship between features.

Definition 4.2. *Given a feature tree $\mathbf{T} = \langle R, \mathcal{L}_T, \mathcal{L}_C, \text{groups}, \text{members}\rangle$, two features are in the parent-child relation if the parent feature owns a group of which the child is a member. This relation is captured by the following equation:*

$$\begin{aligned}&\mathit{child}(p, c : \mathcal{F}) \equiv\\ &\quad (\exists g : \mathcal{G} \bullet g \in \text{groups}(p) \wedge c \in \text{members}(g))\end{aligned}$$

This parent-child relation, and the structure imposed by the definitions of groups and members, enforces a tree structure only on the features that are reachable from the root, and we have formally proven this fact in our PVS theory. (In practice, the tree will not even contain infinite paths.) By not forcing all features to be part of a single tree we can track and reason about features that are not used since, due to the selection mechanism, features not reachable from the root cannot be selected. An addition bonus is that one is not forced to change the feature domain whenever the model is modified.

Definition 4.3. *Given a feature tree $\mathbf{T} = \langle R, \mathcal{L}_T, \mathcal{L}_C, \text{groups}, \text{members}\rangle$, the restriction function of type select : $\mathbf{select}, a : \mathbb{A}$ is a conjunct of the following conditions:*

- *the root of $\mathbf{T}$ must be selected: $\mathit{select}(R)$*

- *whenever a feature f is selected, there must be a path of selected features from the root to f:*

$$\begin{aligned}&\forall f : \mathcal{F}\bullet select(f) \Rightarrow \\ &\quad(\exists f_1, \ldots, f_n\bullet \\ &\qquad(f_1 = R) \wedge (f_n = f) \\ &\qquad\wedge (\forall i \in [1\ldots(n-1)]\bullet child(f_i, f_i + 1)) \\ &\qquad\wedge (\forall i \in [1\ldots n]\bullet select(f_i)))\end{aligned}$$

- *for each group, if its owner is selected, its cardinalities must be satisfied:*

$$\begin{aligned}&\forall p : \mathcal{F}; g : \mathcal{G}\bullet select(p) \wedge g \in \text{groups}(p) \Rightarrow \\ &\quad|\{f : \mathcal{F} \bullet f \in \text{members}(g) \wedge select(f)\}| \in \mathcal{L}_C(g)\end{aligned} \tag{5}$$

We will denote the function taking a feature and returning its restriction function as $restriction$:

$$restriction : \mathbf{TREE} \to (\mathbf{select} \times \mathbb{A} \to \mathbb{B})$$

4.2 Meta-model Properties

Next we present an example of a lemma that relates the concept of a mandatory feature to the cardinality notation [CE00]. By realizing feature relations of "classical" feature models like FODA in our formalism we can "sanity check" our model against the many earlier formalizations of the same constructs. Providing such "native" lemmas and formalisms also provides a natural target for a direct translation of classical feature diagrams into our theory.

The Mandatory Lemma. *If a group g has exactly the admissible cardinality* 1*, and contains exactly one member m, then in any valid configuration that selects the owner of that group, m is selected as well. I.e., m is a mandatory feature.*

$$\begin{aligned}&\forall ft : \mathbf{TREE}, g : \mathcal{G}; m : \mathcal{F}\bullet \\ &\quad(ft.\mathcal{L}_c(g) = \{1\} \wedge ft.\text{members}(g) = \{m\}) \Rightarrow \\ &\quad(\forall s : \mathbf{select}, a : \mathbb{A}\bullet \\ &\qquad(restriction(ft)(s, a) \\ &\qquad\wedge (\exists p : \mathcal{F}\bullet s(p) \wedge g \in ft.\text{groups}(p))) \Rightarrow \\ &\qquad s(m))\end{aligned}$$

Proof sketch. This lemma follows from Definition 4.3's equation 5, imposed by the restriction function. Since the admissible cardinality of the group g, $\mathcal{L}_C(g)$, is exactly 1, and m is the only member of g, m has to be selected whenever the owner of g is selected. We have formalized and proven an analogous lemma for the alternative feature in our theory. □

Our meta-model of feature modeling approaches is designed such that it is easily refined and extended. For example, we might require that all the selected members of a clone group are unique via their attribute values. Any other constraints are then simply expressed as additional conjuncts.

4.3 Model Transformation

Our formalization enables us to reason about the operations on feature models, as illustrated by the following lemma. (Recall that a specialization of a feature model was defined in Section 3.)

The Cardinality Specialization Lemma. *Whenever a new feature tree ft_2 is obtained from an existing feature tree ft_1 by removing some admissible cardinalities of a certain group g, the feature tree ft_2 is a* specialization *of the original tree ft_1. This fact is formalized in the following formula relating two feature trees that differ from one another only in their cardinality function:*

$$\begin{aligned}&\forall ft_1, ft_2 : \mathbf{TREE}, g : \mathcal{G}, \mathcal{L}_C : \mathcal{G} \to \mathcal{P}(\mathbb{N})\bullet \\ &\quad ft_2 = \langle ft_1.R, ft_1.\mathcal{L}_T, \mathcal{L}_C, ft_1.\text{groups}, ft_1.\text{members}\rangle \\ &\quad \mathcal{L}_C(g) \subseteq ft_1.\mathcal{L}_C(g)\wedge \\ &\quad (\forall l : \mathcal{G}\bullet(l \neq g) \Rightarrow ft_2.\mathcal{L}_C(l) = ft_1.\mathcal{L}_C(l))) \Rightarrow \\ &\qquad specialization?(restriction(ft_2), restriction(ft_1))\end{aligned}$$

Proof sketch. We must show that any configuration admissible by ft_2 is also admissible for ft_1. Since the two trees differ only in the set of admissible cardinalities for the group g, only requirement 5 is different in the resulting restriction functions. Let us introduce the following shorthand,

$$S_g \equiv |\{f : \mathcal{F} \bullet f \in \text{members}(g) \wedge select(f)\}|$$

then the proof hinges on the following implication,

$$S_g \in ft_2.\mathcal{L}_C(g) \Rightarrow S_g \in ft_1.\mathcal{L}_C(g)$$

which follows immediately from the precondition that requires $ft_2.\mathcal{L}_C(g) \subseteq ft_1.\mathcal{L}_C(g)$. □

Additionally, in this context we are able to formalize the individual steps of transformations as functions from a more general restriction function to the specialized restriction function. As an example, the following definition introduces the function $assign\text{-}value$, which takes a restriction function r and returns its specialization that requires that a given attribute atr has a given value v. In the following, the type of val requires that the attribute value corresponds to the type of the attribute.

$$\begin{aligned}&assign\text{-}value(r : \mathbf{restr}, f : \mathcal{F}, atr : \texttt{type}(f).ids; \\ &\quad val : \{v : \mathcal{AV} \bullet \texttt{typing}(v) = \texttt{type}(f).att(atr)\}) \equiv \\ &\quad \lambda s : \mathbf{select}, aa : \mathbb{A}\bullet r(s, aa) \wedge (aa(f)(atr) = val)\end{aligned}$$

Properties of such transformations can be investigated as illustrated by the following formula:

$$\forall r : \mathbf{restr}; f : \mathcal{F}; atr : \mathtt{type}(f).ids; \\ val : \{v : \mathcal{AV} \bullet \mathtt{typing}(v) = \mathtt{type}(f).att(atr)\} \bullet \\ specialized?(assign\text{-}value(r, f, atr, val), r)$$

This expressivity provides us with a means of deriving more complicated feature models from simpler ones by specialization composition. By starting from a feature tree, then translating it to a restriction function through the use of the function $restriction$, and then applying specializations on that function, we obtain the desired model. This is schematically illustrated by:

$$\mathrm{fm} = \mathrm{spec}_n(\ldots(\mathrm{spec}_1(\mathrm{restriction}(\mathrm{ft})))\ldots)$$

We are particularly intrigued by the simplicity and richness of this specialization-based (function type subtyping) approach to model refinement.

4.4 Feature Models with Cloning

In Czarnecki et al.'s paper on staged configuration a concrete feature tree of an operating system security profile is used as an example [CBUE02]. Due to space issues, we illustrate here only the topmost nodes of the tree as realized in our formal model.

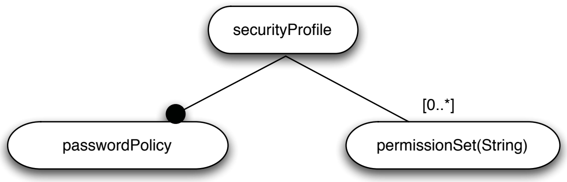

Figure 1. The topmost nodes of the feature tree example from Czarnecki et al. [CHE04b].

A graphical representation of this example feature tree is depicted in Figure 1.

This feature tree is graphically depicted using our formalism in Figure 2. In this diagram style we continue to use ovals to represent features and introduce the use of rectangles to represent groups. A normal feature group is a plain rectangle, (the *passwordPolicyGroup* with a cardinality of 1), and cloned feature groups as a stack of rectangles (see Figure 3). Note that we are *not* introducing a new feature modeling graphical notation, we are only depicting our formal model graphically for the benefit of the reader.

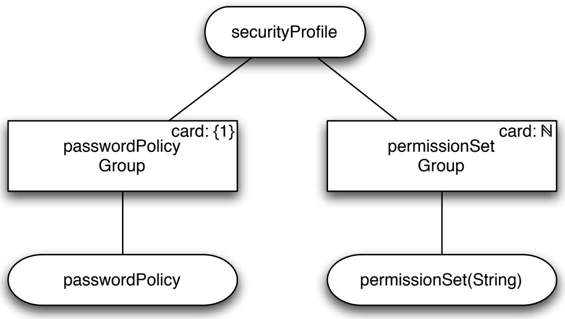

Figure 2. Our representation of the feature tree from Figure 1.

On this example we can illustrate the advantage of using a full HOL description. This model has an unbounded number of configurations as the number of possible assignments is infinite and the *permissionSet* feature can be cloned ad libitum. Thus, to investigate properties, one cannot directly employ a model checker, for example. One such property is illustrated by the following lemma.

Lemma 4.4. *The* passwordPolicy *feature will be included in every valid configuration of the the security profile tree. This fact is stated as a formula quantified over all feature configurations. Please recall that the function* restriction *returns a restriction function for a given feature tree. Let* security-tree *be the tree depicted in Figure 1.*

$$\forall s : \mathbf{select}; aa : \mathbb{A} \bullet \\ restriction(\text{security-tree})(s, aa) \Rightarrow s(\text{passwordPolicy})$$

Proof sketch. This lemma immediately follows from the fact that the root (the *securityProfile feature*) has to be selected in every valid configuration, and from the mandatory lemma (see Section 4.2). □

To show that a this model is consistent, we must provide a configuration and prove that it is valid. In the PVS theory we defined a configuration by selecting the features $\mathrm{securityProfile}$, $\mathrm{passwordPolicy}$, $\mathrm{permissionSet}_0$, and $\mathrm{permissionSet}_1$, where the clones $\mathrm{permissionSet}_0$, $\mathrm{permissionSet}_1$ were assigned arbitrary names. Subsequently we have proven that this configuration is valid.

4.5 Feature Models with Attributes

As our formalized feature meta-model supports attributes, an example including the use of attributes is warranted. This example, focusing on a satellite software case study, is a slightly modified version of an example from Czarnecki et al.'s paper on generative programming [CBUE02].

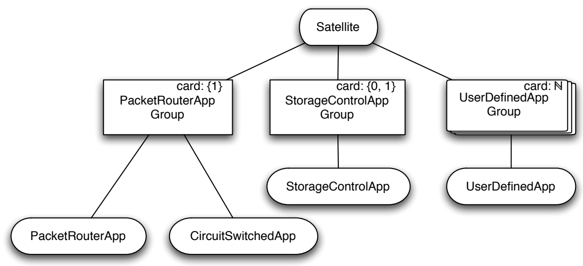

Figure 3. A fragment of the satellite feature model from Czarnecki et al. [CBUE02].

In this example, a fragment of which is summarized in Figure 3, the interface to a satellite system consists of a single mandatory packet router application, an optional storage control application, and a number of applications. Because the hardware resources of a satellite are fixed and precious, we must ensure that any given configuration will fit within those resource bounds. For example, we must ensure that all selected applications will fit in the main memory of the satellite. Such a constraint is easily expressed with attributes.

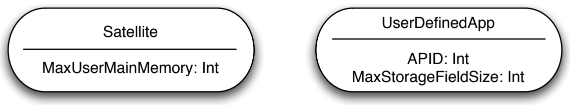

Figure 4. Exploded representation of the features $\mathrm{Satellite}$ and $\mathrm{UserDefinedApp}$ from Figure 3.

The feature $\mathrm{UserDefinedApp}$ has an attribute $\mathrm{MaxStorageFieldSize}$ of type integer. We interpret this attribute to mean the maximum amount of main memory that the given feature can demand of the satellite's software system. Additionally, the feature $\mathrm{Satellite}$ contains the related field $\mathrm{MaxUserMainMemory}$, which indicates a satellite's (fixed, finite) main memory size available for user-defined applications. Both of these attributes are summarized in the "exploded" feature representations in Figure 4.

To formally capture the aforementioned memory constraint, we specify the following restriction function. Let $\mathrm{satellite\text{-}tree}$ be the tree depicted in Figure 3.

$$\begin{aligned}
&\lambda s : \mathbf{select}; aa : \mathbb{A} \bullet \\
&\quad restriction(\mathrm{satellite\text{-}tree}) \wedge \\
&\quad \exists n : \mathbb{N}, \mathrm{sel} : \mathcal{P}(\mathcal{F}) \bullet \\
&\qquad \mathrm{sel} = \{ f : \mathcal{F} \bullet s(f) \\
&\qquad\qquad \wedge f \in \mathrm{members}(\mathrm{UserDefinedAppGroup}) \} \\
&\qquad \wedge |\mathrm{sel}| < n \\
&\qquad \wedge \Bigg(\sum_{c \in \mathrm{sel}} aa(c)(\mathrm{MaxStorageFieldSize}) \\
&\qquad\qquad < aa(\mathrm{Satellite})(\mathrm{MaxUserMainMemory}) \Bigg)
\end{aligned}$$

This constraint defines a new restriction function via *specialization*. The new restriction function specializes the existing restriction function obtained from the diagram by adding the desired constraint as an additional conjunct. The constraint states that the set of the selected clones is finite, and that the sum over the $\mathrm{MaxStorageFieldSize}$ attribute of the selected clones is less than the $\mathrm{MaxUserMainMemory}$ attribute of the $\mathrm{Satellite}$ feature. We have verified valid and invalid configurations against this specialized restriction function in PVS.

This example illustrates the need for two kinds of groups—clone groups and feature groups. To be able to state this example constraint, we need all the members of the $\mathrm{UserDefinedApp}$ to have the attribute $\mathrm{MaxStorageFieldSize}$. In other words, we need for all the members of the $\mathrm{UserDefinedApp}$ group to have the same type. On the other hand, the $\mathrm{PacketRouterApp}$ group, which is a group of alternative features, contains features that are potentially of different type.

The example also suggests how constraints of this kind could be expressed more succinctly. In particular, the introduction of auxiliary functions and constraints for frequently used constructs would be beneficial. It is natural to require that a configuration contains a finite set of features, thus, this constraint should be part of the panoply of the predefined specializations. Another example of a frequently used construct is a function that returns the set of selected features in a given group or a set.

5 Conclusion

Using a HOL formalization of a feature modeling metamodel, many new avenues of investigation are now available. And, while having a mechanical formalization ensures certain meta-theoretical properties about the research (e.g., soundness of the theory, correctness of model refinements and transformations, etc.), there are tradeoffs in demanding this level of rigor. We will first discuss related work that has both influenced this meta-model and that we can model in our theory. Then we discuss the pros and cons of our approach. Finally, we reflect on some potential next research steps.

6 Related Work

Propositional formulas were introduced into the SPL domain by Mannion [Man02]. The principal idea is that variability and commonality, defined by a feature diagram, are translated into a propositional formula where the atoms represent features and the formula is valid if and only if a given configuration is admissible.

This idea has been extensively addressed by Batory as well [Bat05]. In this work a connection is defined between feature diagrams, grammars, and propositional formulas. Czarnecki et al. also introduced the use of context-free grammars as a semantics for their feature model [CHE04b, CHE04a].

Neither of these approaches, propositional formulas or grammars, work well with all models. A semantics given as a propositional formula is not well suited for a meta-model that enables cloning of features, especially in the work of Czarnecki et al. where a feature can be cloned ad libitum. Whereas the grammar-based approaches elegantly model feature cloning, and the structure of the translated diagram is well reflected by the grammar, the full use of cross-cutting dependencies, such as *excludes* and *requires*, is difficult to capture. Most importantly, however, it is not clear how dependencies between attributes should be modeled.

Czarnecki and Kim describe the use of OCL in the context of a feature model to express constraints that are not expressible by the diagram, the so called *additional constraints* [CK05]. Their formalism allows at most one attribute per feature and the type of an attribute is either a primitive type or a reference to another feature. In this context, the feature diagram is translated to UML. In the UML diagram, entities correspond to features and a multiplicity at the aggregate end of every composition is 1, while the multiplicity at the other end is the same as the corresponding cardinality in the feature diagram.

Kim and Czarnecki also addressed the issue of evolution of a feature model in the context of specialization [KC05]. I.e., when a feature model is configured via specialization, and later that feature model evolves, the changes made by the evolution are reflected by the specialized models. The article describes how to cope with a set of evolution changes and specializations. The authors applied the QVT relation language to express the relations between specializations of the original and evolved models.

Benavides et al. apply the techniques of constraint programming to feature models [BTRC05]. The authors extend the feature meta-model with the capability of out-fitting features with attributes and define constraints between these attributes, where the form of constraints is inspired by the work of Streitferdt et al. [SRP03]. The authors provide a translation of the feature model constraints to a constraint programming problem. This translation enables the use a constraint solver for analyzing a given configuration.

This work demonstrates the benefits of translating the constraints between features into a format understood by a reasoning engine. Moreover, the use of constraint programming in the SPL domain appears to be quite promising. The article, however, does not provide the full account of the constraint language used.

Schobbens et al. introduce a generic formalisation of feature models based on the FODA notation[SHT06]. Their meta-model is parameterized by two concepts: a graph type, which can be either a DAG or a tree; and node types, which are sets of boolean functions (e.g., a set of $\mathtt{and}_\mathtt{s}$ for every arity $\mathtt{s} \in \mathbb{N}^+$ forms a node type). Cross-graph constraints are either expressed as *graphical constraints* or by using a *textual constraints language*. In the context of this parameterization, the concept of a feature diagram is defined. Subsequently, the authors define the concepts of a *model* and a *valid model* of a feature diagram. A model of a feature diagram is a subset of its nodes, representing the selected features, and a valid model is such a model that satisfies all the constraints imposed by the diagram. Additionally, the authors examine the expressiveness of their meta-models.

A formalisation of a feature modeling approach using the Z language has also been defined by Sun et al. [SZLW05]. The authors formalize the ODM notation which is a variant of the original FODA notation. The formalisation is realized as a set of relations, each of which defines a different type of the parent-feature relationships (e.g., *alternative*, *mandatory*, etc.). A number of theorems about the meta-model are proven to verify certain desired properties of the definitions. For example, one of these theorems states that if a feature is selected, and its children are *alternative*, then exactly one of its children is selected. Furthermore, the authors realize the formalization in the Alloy tool in a subset of the Z language to automatically reason about feature models.

6.1 Limitations and Strengths

Using a HOL theorem prover is often a double-edged sword. We can express dependencies of arbitrary complexity, as illustrated by the satellite example in Section 4.5. Thus, it is not necessary to introduce new constructs or notations, since the PVS system provides tool support, language, and its semantics. Reasoning at the meta-model level, i.e., statements of the form “for all models ...”, is very difficult or impossible to accomplish in a tool with weaker expressivity (e.g., a tool without higher-order features).

On the other hand, the proofs regarding concrete feature models are unnecessarily tedious. We believe that one of the reasons for this is that the defined feature model imposes too few implicit restrictions. For example, it is possible in our meta-model to have configurations with an infinite number of selected features, or a tree can contain infinite paths. Additionally, as discussed below, we are not using any domain-specific strategies in PVS, thus we are only currently using the "raw" PVS language and proof system with few auxiliary constructs. It is not clear how much these restrictions and constructs will make the reasoning easier.

A scenario we are considering, discussed in more detail in the sequel, is the use a feature modeling tool to describe the desired diagram, automatically produce the PVS representation of the diagram along with the rudimentary proofs, and utilize PVS for dependencies not expressible by a feature diagram. For example, if a feature is selected, a proof has to be given that there is a path to the root of selected features, and constructing such a proof is easily automated.

6.2 Future Work

To support within the Mobius PVE the main metaphor that feature-oriented programmers are used to, we hope to either adopt an existing feature diagramming component, or if necessary, develop our own using GEF. A natural next step is generating GEF, and compiling GEF to PVS theories, a standard syntax for feature graphs. If one does not yet exist, we believe that Graphviz's dot file format is a good candidate given its simplicity, its focus on annotated directed graphs, and its quality tool support.

Once we have a given feature model and proposed selection in PVS, we can check its consistency and satisfiability, as we have shown in this paper. Of course, given that PVS is an interactive theorem prover, such a check, especially when arbitrary constraints on features and attributes are specified, is sometimes a laborious process. In the past we have used several solutions to automate these kinds of standard operations.

First, we can write domain-specific PVS strategies. For a theory as simple as the one presented here, this should be straightforward enough. And, while writing flexible strategies that are robust in the face of theory evolution is a fine art, we have sufficient experience in PVS to accomplish such. Another solution for automation is relying upon a model checker or an SMT solver, both of which are integrated with PVS 4 and the Mobius PVE. By mapping our theory to a model within PVS (in the case of using the model checker), or by relying upon only first-order terms (in the latter case of using an SMT solver), then PVS solves problems like the feature model configuration check automatically and efficiently.

Finally, there are many open questions about feature models that we would like to investigate now that we have a HOL formalization. For example, might feature (meta-)models be more naturally formalized using records and structural subtyping? What other kinds of interesting refinements are there? Should we formalize and rigorously compare other proposed feature models? Does a feature model with some notion of multiple-inheritance make sense?

The full PVS formalization of our theory of feature meta-models, all the discussed examples (and others for which we had no room to discuss), and all proofs is available via our research group's homepage.

7 Acknowledgments

This work is partially supported by Science Foundation Ireland under grant number 03/CE2/I303-1, "LERO: the Irish Software Engineering Research Centre." This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This article reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

References

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In Henk Obbink and Klaus Pohl, editors, *Proceedings of the 9th International Conference, SPLC 2005*, volume 3714 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, September 2005.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In Pastor Oscar and João Falcão e Cunha, editors, *Proceedings of 17th International Conference, (CAiSE 2005)*, volume 3520 of *Lecture Notes in Computer Science*. Springer–Verlag, 2005.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative programming for embedded software: An industrial experience report. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, October 2002.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley Publishing Company, 2000.

[Cha02] Gary J. Chastek, editor. *Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, August 2002.

[CHE04a] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their staged configuration. Technical Report 04–11, Department of Electrical and Computer Engineering, University of Waterloo, Canada, April 2004.

[CHE04b] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In Robert L. Nord, editor, *Proceedings of Third International Conference, SPLC 2004*, volume 3154 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, August 2004.

[CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of International Workshop on Software Factories*, 2005.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Publishing Company, 2002.

[FFB02] Dániel Fey, Róbert Fajta, and András Boros. Feature modeling: A meta-model to enhance usability and usefulness. In Chastek [Cha02].

[KC05] Chang Hwan Peter Kim and Krzysztof Czarnecki. Synchronizing cardinality-based feature models and their specializations. In *Proceedings of European Conference on Model Driven Architecture: Foundations and Applications*, Lecture Notes in Computer Science. Springer–Verlag, 2005.

[KCH$^+$90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA), feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, November 1990.

[Man02] Mike Mannion. Using first-order logic for product line model validation. In Chastek [Cha02].

[PBv05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer–Verlag, 2005.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceeding of 14th IEEE International Requirements Engineering Conference (RE'06)*. IEEE Computer Society, 2006.

[SRP03] Detlef Streitferdt, Matthias Riebisch, and Ilka Philippow. Details of formalized relations in feature models using OCL. In *Proceedings of the 10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003)*. IEEE Computer Society, 2003.

[SZLW05] Jing Sun, Hongyu Zhang, Yuan Fang Li, and Hai Wang. Formal semantics and verification for feature modeling. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*. IEEE Computer Society, 2005.