Abstract Features in Feature Modeling

Thomas Thüm*, Christian Kästner†, Sebastian Erdweg†, and Norbert Siegmund*

*University of Magdeburg, Germany
†Philipps University Marburg, Germany

***Abstract*—A software product line is a set of program variants, typically generated from a common code base. Feature models describe variability in product lines by documenting features and their valid combinations. In product-line engineering, we need to reason about variability and program variants for many different tasks. For example, given a feature model, we might want to determine the number of all valid feature combinations or compute specific feature combinations for testing. However, we found that contemporary reasoning approaches can only reason about feature combinations, not about program variants, because they do not take abstract features into account. Abstract features are features used to structure a feature model that, however, do not have any impact at implementation level. Using existing feature-model reasoning mechanisms for program variants leads to incorrect results. Hence, although abstract features represent domain decisions that do not affect the generation of a program variant. We raise awareness of the problem of abstract features for different kinds of analyses on feature models. We argue that, in order to reason about program variants, abstract features should be made explicit in feature models. We present a technique based on propositional formulas that enables to reason about program variants rather than feature combinations. In practice, our technique can save effort that is caused by considering the same program variant multiple times, for example, in product-line testing.**

***Keywords*-Software product lines, program families, feature modeling, feature model, automated analyses.**

I. Introduction

A *software product line* is a set of software-intensive systems (program variants) that share common code artifacts [1]. The overall idea is to efficiently develop similar programs simultaneously, by systematically reusing development artifacts. Program variants are distinguished in terms of *features*, which are prominent or distinctive user-visible aspects, qualities, or characteristics of a software system [2]. Two variants may have several features in common and differ in other features. In particular, we focus on product-line engineering methods, in which program variants can be *generated* from a common implementation by specifying a selection of features. This includes implementation techniques, such as conditional compilation [3], plug-ins and frameworks, or advanced programming-language mechanisms with aspects [4], feature modules [5], [6], or delta modules [7].

In general, not all combinations of features are useful and result in meaningful program variants. For instance, there might be mutually exclusive features or features that require other features. A variability model specifies all valid *feature combinations* and thus the program variants that can be generated. A common form of variability models are feature models [2], [8], [9]. Feature models can have several representations, e.g., a propositional formula in which each feature belongs to a variable and the formula evaluates to true for all valid combinations [10].

We noticed that not all features in typical feature models are used to distinguish program variants. Some are only used to structure the model and selecting or eliminating them does not make any difference in the generated variant code. We denote such features as *abstract features*. Due to abstract features, the set of *feature combinations* and the set of *program variants* are not equivalent. Multiple valid feature combinations result in the same generated program variant.

There are numerous approaches to reason about valid feature combinations in feature models [9]. However, there are also many interesting questions, for which we want to reason about program variants. For example, for combinatoric testing [11], type checking [12]–[15], or verification [16], we want to select certain sets of program variants or need to reason about the relationship of features in program variants. Similarly, for reasoning about non-functional properties of program variants [17], abstract features are not relevant, but can increase measurement effort significantly. Finally, deleting an abstract feature from the feature model does not change possible program variants and may hence be considered as feature-model refactoring [18]. Contemporary automated analyses of feature models can be used to reason about valid combinations of features, but not to reason about program variants. When using automated reasoning nonetheless, the results are at least inaccurate or inefficient.

As a consequence, we distinguish two semantics: (a) the semantics of feature models as known from literature [19], describing the valid combinations of features and (b) the semantics of program families describing distinct program variants of a product line. According to our experience, both are needed and complementary to each other. However, in order to not redevelop all reasoning mechanisms again for the program-families semantics, we provide a mechanism to translate the program-families semantics in a way that existing reasoning mechanisms for the feature-model seman-

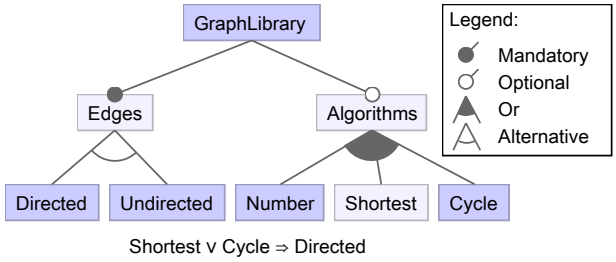

Figure 1. A feature model representing a product line of graph libraries.

Feature Model	Propositional Formula
Optional feature C_i	$C_i \Rightarrow P$
Mandatory feature C_i	$(C_i \Rightarrow P) \wedge (P \Rightarrow C_i)$
Or-group	$P \Leftrightarrow \bigvee_{1 \leq i \leq n} C_i$
Alternative-group	$(P \Leftrightarrow \bigvee_{1 \leq i \leq n} C_i) \wedge \bigwedge_{i<j} (\neg C_i \vee \neg C_j)$

Figure 2. Translation of feature models into propositional formulas. P denotes a compound feature with the subfeatures $C_1, C_2, \ldots, C_n$.

tics can be used. We implemented that translation in our tool framework FeatureIDE [20], developers need to make abstract features explicit during feature modeling.

In summary, we make the following contributions:

- We raise awareness of the problem of abstract features.
- We distinguish two semantics for feature models to reason about both, valid feature combinations and program variants.
- We introduce the notion of abstract features into feature modeling.
- We provide a translation from feature-model semantics to program-family semantics based on propositional formulas, so that existing reasoning mechanism for feature models can be used for reasoning about program variants.
- We provide an open-source implementation as part of FeatureIDE.[1]

II. Feature Models

A *feature model* specifies the features of a product line and the valid combinations thereof [2]. Feature models have a hierarchical structure: each model has one root feature and every feature can have further subfeatures. The subfeatures of a feature can either be mandatory, optional, or can be in an Or-group or an Alternative-group [8], [21]. A feature is called *compound feature* if it has subfeatures and *primitive feature* otherwise [10].

We give an example in Figure 1. The feature model represents a product line of graph libraries and is a simplified version of a real existing feature model designed by others.[2] Variants of the graph library always support edges which are either directed or undirected. A graph library may also provide algorithms, such as assigning a consecutive number to all nodes, calculating the shortest path between two given nodes, or detecting cycles in graphs.

Besides the hierarchical structure, many feature modeling notations also allow cross-tree constraints. A *cross-tree constraint* is an additional constraint (often an arbitrary propositional formula) on the features of the feature model [10]. If a feature model contains multiple cross-tree constraints, all of them must be fulfilled. In our example, the cycle detection and calculation of a shortest path require directed graphs, i.e., those algorithms are not available for graphs with undirected edges.

A feature model can be translated into a single *propositional formula* which evaluates to true if and only if the combination of features is valid. It is constructed by conjoining (a) the propositional formula for each construct in the feature model (see Figure 2), (b) all cross-tree constraints, and (c) a formula requiring the root feature [10]. The resulting propositional formula describes the *semantics of feature models*, i.e., the valid combinations of features. In the following, we give the propositional formula for our example feature model.

$$(E \Rightarrow G) \wedge (G \Rightarrow E) \wedge (A \Rightarrow G) \wedge (E \Leftrightarrow D \vee U) \wedge (\neg D \vee \neg U) \wedge (A \Leftrightarrow N \vee S \vee C) \wedge (S \vee C \Rightarrow D) \wedge G$$

In addition, a feature model can be represented as a set of sets of features [19]. Given a set of features F, a *configuration* (feature combination) C is defined as $C \subseteq F$. Then, a feature model can be specified by a set S of configurations. A configuration C is called *valid* if and only if $C \in S$.

There are many different notations and extensions of variability models [9], [19]. Our approach is general in that it can deal with all models that can be translated into propositional formulas, which is possible for most notations and extensions [22].

III. Mapping Features to Implementations

To reason about program variants, we briefly need to introduce typical implementation mechanisms. We focus only on mechanisms, which, for a given configuration, can generate a program variant from a common implementation automatically. To allow this generation, there is typically some more or less complex mapping between features (problem space) and implementation artifacts (solution space).

In the simplest case, we can map each (non-abstract) feature to a single code unit. For example, we can map a feature to a plug-in for a framework, to an aspect [4], a feature module [5], [6], or a delta module [7]. In Figure 3, we excerpt an implementation using AHEAD-style feature

[1] http://fosd.net/featureide
[2] http://fosd.net/featurehouse

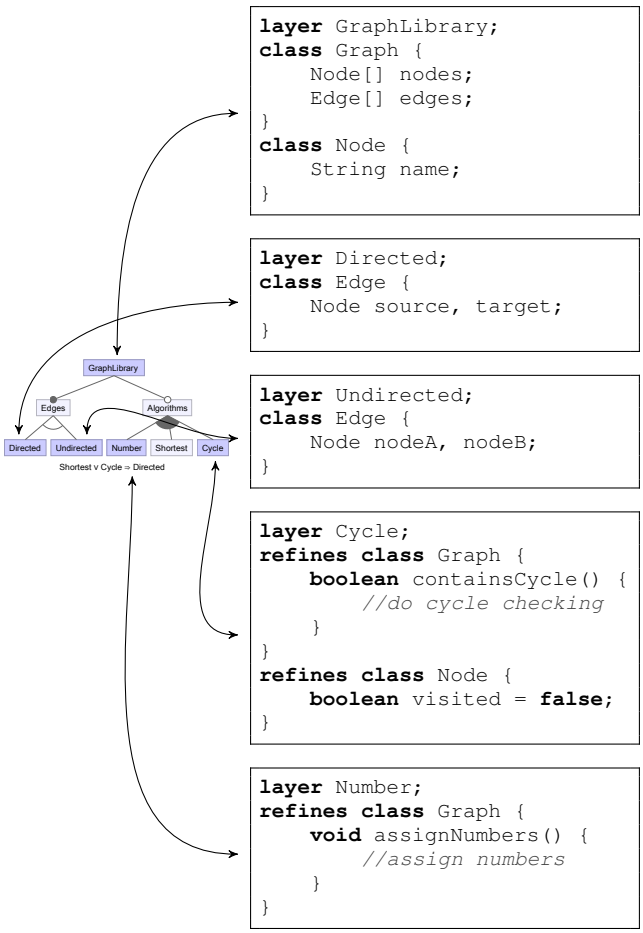

Figure 3. A 1:1 mapping between features and feature modules using feature-oriented programming.

modules, in which each non-abstract feature is mapped to exactly one implementation module containing classes or class refinements. A variant is generated by assembling the modules that correspond to selected features and composing (or weaving) the corresponding code fragments. Proponents of feature-oriented software development argue that this simple 1:1 mapping is a strength in understanding and structuring the product-line implementation [23].

In more complex scenarios or with other implementation techniques, a simple 1:1 mapping may not be sufficient. For example, when variability is implemented with conditional compilation such as #ifdef directives of the C preprocessor, a single feature can be mapped to multiple code fragments. In addition, the inclusion of a single code fragment may be influenced by multiple features. In Figure 4, we show such a mapping for an implementation of the graph library using conditional compilation. Again, a variant is generated by assembling the code fragments that correspond to a configuration. Besides conditional compilation, many product-line mechanisms and tools, such as pure::variants [24], CIDE [25], FeatureMapper [26], or application conditions in

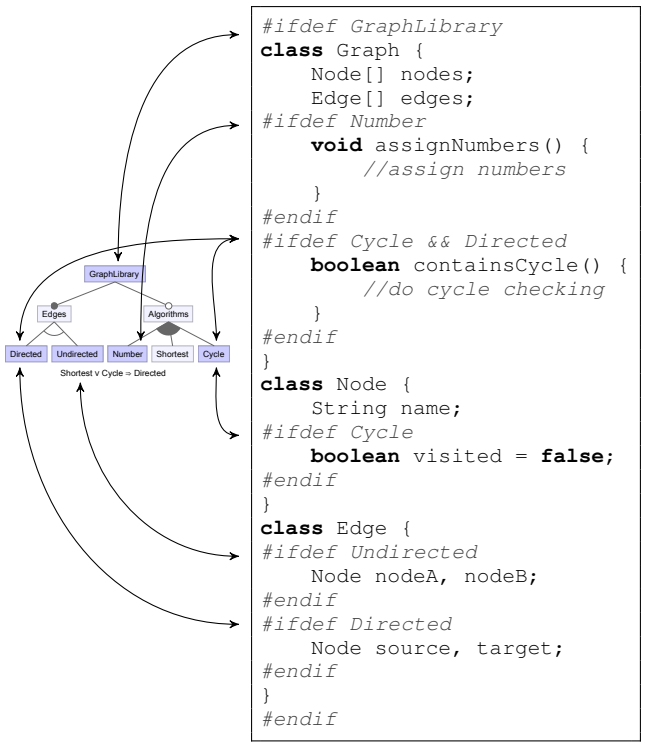

Figure 4. An n:m mapping between features and preprocessor statements.

DeltaJ [7], allow complex mappings between features and implementation artifacts.

Finally, in some implementation settings, developers can even script arbitrary Turing-complete mappings. Such configuration scripts are not accessible to automated reasoning, unless heavyweight solutions like symbolic execution [27] are employed. Examples are build systems and Gears [28].

In all cases, we can see that certain features do not affect the implementation (in complex mappings, it might not be easy to recognize automatically though). For example, selecting or not selecting *Shortest* will not affect generated program variants, because it is not (yet) mapped to implementation artifacts.

IV. Reasoning about Program Variants

When implementing a software product line, it is often necessary to reason about all valid program variants. We give some example scenarios:

- We have implemented a software product line and want to test [11], type check [12]–[15], [29], or verify all program variants [16].
- We want to detect dead code, i.e., code fragments that are never used in any program variant [30], [31].
- When changing the feature model, we want to determine whether there are any changes to program variants (i.e., whether feature-model change is a refactoring) [18], [22].

- We want to estimate non-functional properties per feature or per variant (such as code size, energy consumption, or response time), in which case we are, of course, only interested in features that actually affect program variants [17].

For many of these questions, we may be tempted to use automated analyses of feature models, that has been broadly investigated in literature [9]. A typical strategy is to translate a feature model into a propositional formula and subsequently use off-the-shelf solvers, such as satisfiability solvers, BDDs, or CSP solvers, to reason about valid configurations [9], [10].

Unfortunately, existing automated analyses for feature models will usually yield inaccurate results, when trying to answer questions about program variants, because the same program variant may be generated given several distinct configurations. In the presence of abstract features, we may consider identical program variants multiple times, leading, for example, to a higher number of program variants required for combinatoric testing and proofs during verification that are more complex than actually necessary.

In Figure 5, we illustrate the mismatch between configurations and program variants using our running example. At the left side, all valid configurations are enumerated, in which initial letters are used to identify features. Arrows indicate the generation of program variants from configurations. At the right side, the according program variants are described as sets of implementation artifacts. For simplicity, we denote the code artifacts belonging to a certain feature with lowercase letters. For instance, $\{G, E, D\}$ and $\{G, E, D, A, S\}$ produce identical programs as A and S are not mapped to any implementation artifact. There are ten different valid configurations, but only six different program variants.

A. Making Abstract Features Explicit

The mismatch between configurations and program variants is primarily caused by abstract features.[3]

> *Definition*: We define a feature as *abstract*, if and only if it is not mapped to any implementation artifacts. We call all other features non-abstract or *concrete*, i.e., a concrete feature is mapped to at least one implementation artifact.

We require the product-line developer to specify for each feature whether it is abstract or not. For some implementation mechanisms, we could deduce which features are abstract by analyzing the implementation and the existing mapping between features and code, especially when simple one-to-one mappings are used as in Figure 3. However, as explained in Section III, in general such analysis is difficult. We would need to invest into different analysis methods for different kinds of mappings (if possible at all). Hence, we advocate a simpler approach, in which developers manually specify which features are abstract. To support developers in the general cases nonetheless, we provide a checking mechanisms for known and simple kinds of mappings in our implementation, which issues a warning if an abstract feature is mapped to implementation artifacts or a concrete feature has no mapping.

[3]There may be cases, in which two features are mapped to identical code fragments. This would lead to similar mismatch between configurations and program variants as abstract features. However, in our experience such cases are rare and negligible for most kinds of analysis, whereas abstract features are pervasive in most feature models. Hence, we focus on abstract features.

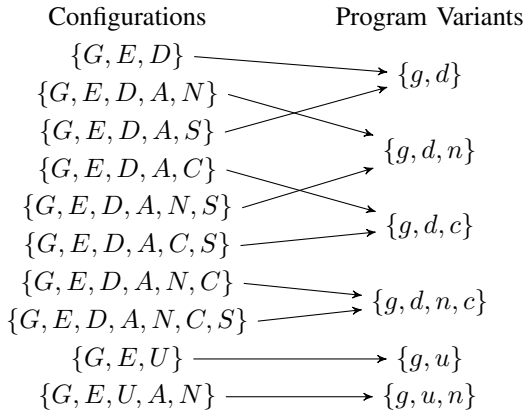

Figure 5. Generating a program variant given a valid configuration corresponds to a surjective function, i.e., multiple configurations may be mapped to the same program variant.

In feature models, such as in Figure 1, we depict abstract features with a lighter background color. Two configurations that differ only in abstract features correspond to the same program variant.

We are unaware of any prior explicit distinction between abstract and concrete features. Previous formalizations of the semantics of feature models do not take abstract and concrete features into account [19], [32]–[34]. Nevertheless, abstract features have been used implicitly (cf. also related work in Section VII) and have actually been a cause of confusion:

- In existing product-line implementations, often, some features are not mapped to code. However, we have seen very different mappings, and there does not seem to be a general pattern of which features are abstract or concrete. Sometimes the root feature and all or some compound features were mapped to implementation artifacts, sometimes they were not; in some cases, some primitive features were abstract, in others, primitive features were always concrete.
- Several authors acknowledge that not all features of a product line are necessarily mapped to code, but they do not specify which features are abstract and do not discuss implications for automated reasoning [30], [35], [36].
- Some tools actually prevent users from mapping

compound features to implementation artifacts (e.g., GUIDSL from the AHEAD tool suite [5] and XConfig [30]), whereas other tools allow arbitrary mappings to any features, so that root, compound, and primitive features can potentially all be abstract or not (e.g., pure::variants [24]).

- In discussions with product-line implementers, we heard different and contradictory assumptions on which features can or should be mapped to code. Some experts have strong opinions that certain features have to or must not be abstract, although they could not point us an existing specification. Interestingly, also in reviews for a prior submission of this paper, we received strong, but contradicting opinions as feedback. One reviewer insisted that the root of a feature model is always assumed to be abstract, whereas another reviewer found it natural that all features are abstract if and only if they are compound.

We conclude that abstract features are relevant for feature modeling, but in current notations, they are assumed only implicitly and not clearly specified. We argue that whether a feature is abstract or concrete should be part of the modeling notation and semantics or at least part of the tool infrastructure that is used for automated reasoning about program variants.

B. Abstract Features in Practice

To further illustrate the impact of abstract features and the mismatch between configurations and program variants, we take a closer look at four case studies and compare valid configurations with program variants. We selected four publicly available feature models with corresponding implementations.[4] The selected product lines were developed by disjunct teams independent of this research project. We manually explored mappings between feature model and implementation artifacts to identify abstract features.

As shown in Figure 6, we found between 18 and 66 percent abstract features in the analyzed software product lines. Abstract features were (a) used to structure the feature model by grouping other features, (b) required to work around limitations of the used tool infrastructure (indicated by meaningless names), or (c) used for features not yet implemented. In three out of four cases, we found a mismatch between configurations and program variants. The strong mismatch in Devolution results from the fact that many features were modeled but have not been implemented (yet).

These numbers show that reasoning about program variants using traditional reasoning techniques for feature configurations may lead to significantly distorted results. Hence, we investigate how to overcome this mismatch with a special treatment of abstract features during analysis.

[4]Unfortunately, for most existing feature model examples, such as those in the SPLOT repository, no mappings to implementations are available and abstract features are not documented [37].

	GPL	SensorNetwork	Devolution	DesktopSearcher
# Features	38	28	32	22
# Concrete primitive features	20	16	11	14
# Concrete compound features	0	7	0	2
# Abstract primitive features	7	1	9	0
# Abstract compound features	11	4	12	6
# Configurations	156	3240	19 656	462
# Program variants	126	1560	84	462

Figure 6. Statistics on how concrete and abstract features are used and mismatch between valid configurations and program variants in four case studies.

V. Semantics of Program Families

To support analyses on program variants, we distinguish between two semantics.

- The *semantics of feature models* describes valid configurations and can be used to derive statistics about feature models and to detect inconsistencies in feature models, e.g., dead features which are not contained in any valid configuration or false optional features which are declared as optional but actually required in all valid configurations due to cross-tree constraints [9]. This semantics is well specified in literature [19], [32]–[34].
- The *semantics of program families* describes all program variants that can be generated using valid configurations. This semantics is necessary when reasoning about program variants, for example, to perform pairwise testing of features [11], or to compute non-functional properties for features [17]. In all these cases, redundant computations caused by the mismatch between configurations and program variants should be avoided by applying the semantics of program families.

We argue, that both semantics are needed and complementary. For every problem, one has to decide which semantics should be applied and used.

A. Eliminating Abstract Features

Fortunately, we can transform the semantics of feature models into the semantics of program families if abstract features are explicitly marked in the feature model. In a nutshell, we can then derive program variants basically by removing all abstract features from configurations, as indicated in Figure 5.

However, in general, it is not feasible to first compute the set of all configurations and then remove abstract features to gain the set of all program variants, because the number of configurations grows up-to exponentially in the number of features. Hence, we propose a mechanism to transform a propositional formula specifying all valid configurations

into a propositional formula specifying the set of distinct program variants.

Technically, we eliminate all abstract features in the propositional formula describing the feature model (semantics of feature models). The result is a propositional formula that contains only concrete features (semantics of program families). To eliminate an abstract feature A from a propositional formula p, we substitute A by its possible values (selected or not selected): We create a new propositional formula $p' = p[A \rightarrow \mathit{true}] \vee p[A \rightarrow \mathit{false}]$, where $p[A \rightarrow X]$ denotes the formula resulting when replacing all occurrences of A in p with X.[5] This process can be repeated until all abstract features are eliminated. The final propositional formula evaluates to true for all combinations of concrete features that are valid according to the feature model, which uniquely describe all program variants of the product line.

B. Optimizations

The presented transformation does not scale well as is, because with every elimination of an abstract feature, we double the size of the propositional formula. However, there are several optimizations that can be applied to prevent exponential explosion for common feature models.

Since the propositional formula describing a feature model is usually in conjunctive normal form or close to conjunctive normal form, optimizations based on boolean algebra can be applied. We only need to substitute an abstract feature in those clauses that contain the feature using commutativity and distributivity, so that duplication occurs locally. In addition, we can apply straight forward simplifications, such as $A \wedge \mathit{true} \equiv A$ or $A \Rightarrow \mathit{true} \equiv \mathit{true}$. In Figure 7, we illustrate the removal of abstract features in our running example using these optimizations.

A further helpful optimization is to replace abstract features by a definition in terms of concrete subfeatures. That is, variables representing an abstract feature are substituted by a disjunction of its subfeatures. We developed this technique in previous work, but in general, there is not always a definition for abstract features (especially in the presence of cross-tree constraints and abstract primitive features) [18], [22]. Still, whenever possible, we use this substitution instead of the general pattern, and only fall back to the general pattern when required.

C. Implementation

We have implemented the possibility to explicitly mark abstract features and the transformation from feature-model semantics to program-family semantics in FeatureIDE [18], [20]. FeatureIDE provides a feature model editor and some reasoning facilities (e.g., reasoning about feature model edits, determining and counting valid configurations) based on a translation of feature models to propositional formulas and using a satisfiability solver. FeatureIDE is open source and available at http://fosd.net/featureide.

[5]Some notations of propositional formulas do not allow *true* and *false* in formulas. In this case, we introduce two fresh variables T and F and substitute as follows: $p' = (p[A \rightarrow T] \vee p[A \rightarrow F]) \wedge T \wedge \neg F$.

$$
\begin{aligned}
p_0 &= (E \Rightarrow G) \wedge (G \Rightarrow E) \wedge (A \Rightarrow G) \\
&\quad \wedge (E \Leftrightarrow D \vee U) \ \wedge (\neg D \vee \neg U) \\
&\quad \wedge (A \Leftrightarrow N \vee S \vee C) \wedge (S \vee C \Rightarrow D) \wedge G \\
&\quad \textit{// elimination of E:} \\
p_1 &= \big((\mathit{true} \Rightarrow G) \wedge (G \Rightarrow \mathit{true}) \wedge (\mathit{true} \Leftrightarrow D \vee U) \\
&\quad \vee (\mathit{false} \Rightarrow G) \wedge (G \Rightarrow \mathit{false}) \wedge (\mathit{false} \Leftrightarrow D \vee U)\big) \\
&\quad \wedge (A \Rightarrow G) \ \wedge (\neg D \vee \neg U) \\
&\quad \wedge (A \Leftrightarrow N \vee S \vee C) \wedge (S \vee C \Rightarrow D) \wedge G \\
&\quad \textit{// simplifications:} \\
p_1' &= (D \vee U) \wedge (A \Rightarrow G) \wedge (\neg D \vee \neg U) \\
&\quad \wedge (A \Leftrightarrow N \vee S \vee C) \wedge (S \vee C \Rightarrow D) \wedge G \\
&\quad \textit{// elimination of A and simplifications:} \\
p_2 &= (D \vee U) \wedge (\neg D \vee \neg U) \wedge (S \vee C \Rightarrow D) \wedge G \\
&\quad \textit{// elimination of S and simplifications:} \\
p_3 &= (C \Rightarrow D) \wedge (D \vee U) \wedge (\neg D \vee \neg U) \wedge G
\end{aligned}
$$

Figure 7. Transforming the propositional formula describing the valid configurations into a propositional formula describing the program variants of our running example.

In FeatureIDE, developers can explicitly mark features as abstract, which are represented using a different background color (all feature models in this paper are in fact exported from FeatureIDE). FeatureIDE also supports a number of different implementation techniques and mappings, and can provide warnings when a feature marked as abstract actually maps to implementations or vice versa.

Based on these markings, FeatureIDE implements the translation into program-family semantics, so that reasoning is possible both about configurations and program families. For example, we may classify a feature model edit as refactoring or specialization depending on which semantics is used. Again, there are reasons for both semantics and depending on the task both might be needed. Our implementation in FeatureIDE offers the flexibility to select which meaning is required for the task at hand.

Regarding performance, due to the optimizations outlined above, our current implementation can easily translate all feature models from Figure 6 in less than a second. Therefore, we omit further performance measures.

VI. Discussion

To support the semantics of program families, we propose that designers of feature models should explicitly specify for each feature whether it is abstract. Instead of leaving the decision and full flexibility to the developer, we could also enforce restrictions on possible notations, some of which

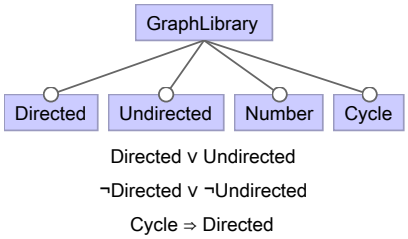

Figure 8. A feature model without abstract features describing the same variants of graph libraries as the feature model in Figure 1.

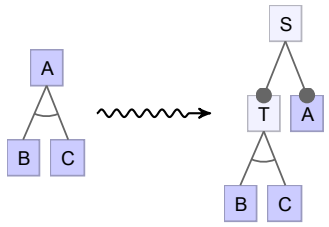

Figure 9. Converting concrete compound features to concrete primitive features when forcing that all compound features are abstract. The feature *A* is a concrete compound feature. We create two new abstract features *S* and *T*. In the translated feature model, all compound features are abstract.

seem to be implicitly assumed by some developers anyway (cf. Section IV-A). In the following, we defend our decision for more flexibility instead of explicit conventions.

A. Avoidance of Abstract Features

We could simply avoid abstract features by forcing that every feature in a feature model is mapped to implementation artifacts. In this case, the semantics of feature models and program families would be identical and no further transformations of propositional formulas would be necessary.

However, our experience and case studies in Figure 6 have shown that abstract features (especially abstract compound features) are useful instruments for structuring a feature model and increasing the comprehension of the variability and provided functionality of a product line.

Although we can always avoid abstract features by translating their structures to cross-tree constraints (which is in fact similar to what our translation does), the resulting formulas or feature models usually loose readability. For example, the feature model in Figure 8 is equivalent to the one in Figure 1 (with respect to program-family semantics), but it is much harder to recognize which possible algorithms on graphs are provided, since the abstract feature *Algorithms* was eliminated. Furthermore, avoiding abstract features may lead to additional cross-tree constraints. In our example, it is not obvious that Directed and Undirected are alternative features, which is stated by the first and the second cross-tree constraints. Both, the removed groups and the additional constraints make the feature models harder to understand.

Additionally, given a feature model and a certain mapping to implementation artifacts, we would need to remove all abstract features manually from the feature model, before we can reason about program variants. Manually removing abstract features requires non-trivial adjustments of cross-tree constraints containing the feature to remove. Here, tool support is necessary and the presented semantics for program families can even help when refactoring feature models.

Furthermore, there is a minor conceptual issue regarding the root feature. If we force that the root feature must be concrete, we cannot express all meaningful software product lines. The reason is that a concrete root feature is a core feature, i.e., a feature contained in every program variant. Thus, if the root feature must be concrete, we cannot express product lines without core features. Hence, even advocates of having only concrete features typically allow (or require) the root feature to be abstract.

In summary, we believe that abstract features are essential for readability of feature models and can provide further structure that can be helpful for stakeholders.

B. All Compound Features are Abstract

As argued above, abstract features can and should not be avoided entirely. But then, instead of specifying for every single feature whether it is abstract or concrete, we could enforce a pattern, for example, specifying that all compound features (i.e., feature with subfeatures) are abstract and all primitive features are concrete. This way, the notation specifies what is abstract and not the developer. This idea is already realized in some tools such as GUIDSL [5], [10] and XConfig [38], [39]. Requiring that all compound features are abstract comes with disadvantages though, which we experienced when we committed to this pattern in earlier versions of FeatureIDE.

First, translating feature models with concrete compound features into a feature model in which all compound features are abstract is generally possible, however, this may reduce readability. We can make all compound features abstract by adding abstract features and moving all subfeatures of the compound feature to the newly created abstract feature as illustrated in Figure 9. Using this conversion, we get many new abstract features that may complicate the feature model more than necessary.

Second, this pattern restricts design choices for graphical feature-model editors. In earlier versions, FeatureIDE's graphical editor was based on the GUIDSL format and, hence, required that all compound features are abstract. To not break the conventions, we cannot allow to add subfeatures to concrete primitive features (the feature would become an abstract compound feature) and we cannot allow to add an abstract feature that does not have subfeatures yet. The solution in FeatureIDE was to provide an operation "create compound feature above" to add abstract features as

parent to an existing feature. In practice, this turned out as counterintuitive and inconvenient. We experienced usability issues of different implementations in four years of teaching with this model, when observing our students while using FeatureIDE.

We tried two options, but none of them was satisfying from a usability standpoint. In the first year, we allowed to create new subfeatures for every feature, with the effect that adding a subfeature can make a concrete feature abstract, so code written for that feature cannot be used anymore. In the second and third year, we ensured that only compound features can get new subfeatures and new compound features would be created *above* a feature. The way of creating features was far from natural thinking and it required to rename features when a compound feature needed implementation artifacts, which was not expected before.

Only when we allowed students to specify which features are abstract in our latest version as proposed in this paper (fourth year of teaching with FeatureIDE), we observed that students could intuitively use operations to modify feature models. In addition, warnings about abstract features with a mapping to code and about missing mappings from concrete features to code turned out helpful to force users to keep information about abstract features up to date (which can be clearly seen by comparing feature models and implementations created in these four years).

In summary, we conclude that enforcing patterns would be possible, but from a usability point of view, we believe that we should extend the feature modeling notation with abstract features instead of some external specifications how to understand existing models.

VII. Related Work

Implicit usage of abstract features: In related work, abstract features were used only implicitly. Batory et al. note that some features are "empty", because no code needs to be written to implement that functionality [35]. They give an example, but do not state for every feature whether it is abstract or not. White et al. argue that the developer's desire to provide a well-structured hierarchy leads to the fact that mostly only primitive features affect the implementation of a product or consume resources [36]. Sincero et al. presented a feature model with a compound feature *Search* and its alternative subfeatures *DFS*, *BFS*, and *None* [30]; we assume that the feature *None* is an abstract feature and maybe even *Search* might not come with any implementation artifacts.

We are not aware of any feature modeling approach that makes abstract features explicit. Nevertheless, we found tools that support abstract features in some form. The feature-modeling language GUIDSL and the corresponding reasoning tool (part of the AHEAD tool suite) encodes the convention that every compound feature is abstract [5], [10]. Therefore, code can only be assigned to primitive features. The reason for this choice in GUIDSL is that a feature model is represented as a grammar. Compound features are mapped to non-terminal symbols and primitive features are modeled as terminal symbols. A valid word according to the grammar corresponds to a program variant and since non-terminal symbols cannot occur in a word, all compound features are abstract.

The Linux kernel is a software product line implemented using C's preprocessor [3], [30], [39]. LinuxKernelConf can be seen as a feature modeling language [30]. It was originally intended to improve the process of deriving valid Linux kernel variants, but it can also be applied to other projects. The tool XConfig is used to configure valid variants using a feature model in the language LinuxKernelConf. The language basically consists of entries with a name and some constraints to other features. Entries either define a configuration option or just help to organize other entries [38]. We and others [30], [39] argue that entries are equivalent to features, since a LinuxKernelConf document specifies the valid combinations of entries. Furthermore, an entry that gives no configuration option and is only used to organize other entries is equivalent to an abstract feature according to our notation. As for GUIDSL the LinuxKernelConf language assumes that every compound feature is abstract, i.e., entries that have other entries as children do not provide configuration options.

The tool SPL Conqueror uses an integrated product-line model that distinguishes between domain features and code units implementing the domain features [17]. Each domain feature may be mapped to a code unit or not. According to our notation, domain features which are not mapped to a code unit are abstract. SPL Conqueror is primarily used to trace non-functional properties from the implementation to the domain features to support product derivation [17].

Finally, some researchers have noticed the mismatch between valid feature configurations and program variants for specific mappings. This was typically detected when analyzing implementations containing variability [12], [29], [31], [32]. However, these approaches are always restricted to specific implementation methods, for example explicit machine-readable mapping between feature model and implementation model [32], one-to-one mappings between features and modules [29], or conditional-compilation based implementations [12], [31]. In general, mappings may be complex and Turing-complete, so that automated reasoning is not always feasible or requires heavyweight analysis techniques. In contrast, our abstract-feature solution is lightweight, can be used independently of the mapping mechanisms, and can still be used to reason about the set of distinct program variants.

Semantics and other variability models: The semantics of feature models was formalized to avoid ambiguities [32]–[34]. To the best of our knowledge, no formal semantics supports abstract features appropriately or discusses program families. For instance, Bontemps et al. presented a formal

semantics, in which a feature model is a tuple [34]; this semantics can easily be extended by the set of abstract features, which is a subset of all features.

In this paper, we discussed only FODA-style feature models [2], [8] and their semantics. However, there are various extensions and alternative variability modeling notations [9]. For example, Riebisch et al. introduced group cardinalities in feature models [40], which specify a lower bound n and an upper bound m of subfeatures that need to be selected if the parent feature is selected. As many other extensions, group cardinalities can easily be expressed using cross-tree constraints [22], [41] and thus can be transformed into propositional formulas, meaning that our approach can be applied to such models as well.

Finally, our solution is based on, but not restricted to propositional formulas. Feature models can be converted to several solvers and logics, e.g., constraint satisfaction solvers [42] or higher-order logic [43]. The need for abstract features and eliminating abstract features for reasoning about program variants remains.

Extracting feature models: Czarnecki and Wasowski proposed an algorithm to extract a feature model from a propositional formula [44], [45]. Given the formula they calculate an implication hypergraph, while the possible output feature models correspond exactly to minimum spanning trees. In case that there is no minimum spanning tree, we cannot construct a feature model without abstract features [44]. If abstract features are allowed, one could add one or more abstract features and additional edges at the implication hypergraph such that there exists a minimum spanning tree for it. The resulting algorithm could extract a feature model for every given propositional formula.

VIII. Conclusion and Future Work

We discussed the possible mappings between features and implementation artifacts using different product-line implementation techniques. A peculiarity is that some features may not be mapped to any implementation artifacts and in case studies we found surprisingly many such features, which we call abstract features. A problem with abstract features is that the semantics of feature models does not allow to reason properly about program variants. As a result, when requesting program variants, we may get certain program variants several times. To prevent from possibly redundant calculations when testing or verifying product lines, we propose to make abstract features explicit in feature models and show how a propositional formula can be retrieved describing the set of distinct program variants.

In future work, we will evaluate how efficient such propositional formulas can be retrieved for very large feature models with some or many abstract features. According to our experience, the specification of which features are abstract and which are concrete should be verified by the tool (and can sometimes even be extracted from implementation and mapping). We already implemented mechanisms for one-to-one mappings in feature-oriented programming, but, further research is required to automatically extract information from other and more complex mappings.

Acknowledgment

We like to thank Don Batory, Marko Rosenmüller, Tillman Rendel, and Sebastian Henneberg for interesting discussions on abstract features. Many thanks to the students team committed to improve FeatureIDE, especially Fabian Benduhn and Jens Meinicke. Christian Kästner and Sebastian Erdweg are supported by the ERC Starting Grant No. 203099. Norbert Siegmund is funded by the German ministry of education and research (BMBF), Nb. 01IM10002B. FeatureIDE is sponsored by the Metop Research Center, Magdeburg.

References

[1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[3] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2010, pp. 105–114.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 2072. Springer, 2001, pp. 327–354.

[5] D. Batory, "AHEAD Tool Suite," Website, available online at http://userweb.cs.utexas.edu/users/schwartz/ATS/fopdocs/; visited on June 10th, 2011.

[6] S. Apel, C. Kästner, and C. Lengauer, "FeatureHouse: Language-Independent, Automated Software Composition," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 221–231.

[7] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-Oriented Programming of Software Product Lines," in *Proc. Int'l Software Product Line Conference (SPLC)*, ser. LNCS, vol. 6287. Springer, 2010, pp. 77–91.

[8] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, 2000.

[9] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Information Systems*, vol. 35, no. 6, pp. 615–708, 2010.

[10] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proc. Int'l Software Product Line Conference (SPLC)*, ser. LNCS, vol. 3714. Springer, 2005, pp. 7–20.

[11] S. Oster, F. Markert, and P. Ritter, "Automated Incremental Pairwise Testing of Software Product Lines," in *Proc. Int'l Software Product Line Conference (SPLC)*, ser. LNCS, vol. 6287. Springer, 2010, pp. 196–210.

[12] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type Checking Annotation-Based Product Lines," *Transactions on Software Engineering and Methodology (TOSEM)*, 2011, to appear.

[13] T. Thüm, "A Machine-Checked Proof for a Product-Line–Aware Type System," Master's thesis, University of Magdeburg, Germany, 2010.

[14] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Type Safety for Feature-Oriented Product Lines," *Automated Softw. Eng. (ASE)*, vol. 17, pp. 251–300, 2010.

[15] I. Schaefer, L. Bettini, and F. Damiani, "Compositional Type-Checking for Delta-Oriented Programming," in *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 43–56.

[16] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel, "Proof Composition for Deductive Verification of Software Product Lines," in *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*, 2011, to appear.

[17] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov, "Scalable Prediction of Non-functional Properties in Software Product Lines," in *Proc. Int'l Software Product Line Conference (SPLC)*, 2011, to appear.

[18] T. Thüm, D. Batory, and C. Kästner, "Reasoning about Edits to Feature Models," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 254–264.

[19] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic Semantics of Feature Diagrams," *Computer Networks*, vol. 51, no. 2, pp. 456–479, 2007.

[20] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A Tool Framework for Feature-Oriented Software Development," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 611–614, formal demonstration paper.

[21] M. L. Griss, J. Favaro, and M. d' Alessandro, "Integrating Feature Modeling with the RSEB," in *Proc. Int'l Conf. Software Reuse (ICSR)*. IEEE Computer Society, 1998, pp. 76–85.

[22] T. Thüm, "Reasoning about Feature Model Edits," Bachelor's thesis, University of Magdeburg, Germany, 2008.

[23] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *J. of Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, 2009.

[24] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability Management with Feature Models," *Science of Computer Programming (SCP)*, vol. 53, no. 3, pp. 333–352, 2004.

[25] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2008, pp. 311–320.

[26] F. Heidenreich, J. Kopcsek, and C. Wende, "FeatureMapper: Mapping Features to Models," in *Companion Int'l Conf. Software Engineering (ICSE)*. ACM, 2008, pp. 943–944, informal demonstration paper.

[27] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, pp. 385–394, 1976.

[28] C. W. Krueger, "Easing the Transition to Software Mass Customization," in *Proc. Int'l Workshop on Software Product-Family Engineering (PFE)*, ser. LNCS, vol. 2290. Springer, 2002, pp. 282–293.

[29] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe Composition of Product Lines," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2007, pp. 95–104.

[30] J. Sincero and W. Schröder-Preikschat, "The Linux Kernel Configurator as a Feature Modeling Tool," in *Proc. Int'l Software Product Line Conference (SPLC)*. Lero Int. Science Centre, University of Limerick, Ireland, 2008, pp. 257–260.

[31] R. Tartler, J. Sincero, D. Lohmann, and W. Schröder-Preikschat, "Efficient Extraction and Analysis of Preprocessor-Based Variability," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2010, pp. 33–42.

[32] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval, "Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis," in *Proc. Int'l Conf. Requirements Engineering (RE)*. IEEE Computer Society, 2007, pp. 243–253.

[33] J. Sun, H. Zhang, Y. F. Li, and H. Wang, "Formal Semantics and Verification for Feature Modeling," in *Proc. Int'l Conf. Engineering of Complex Computer Systems (ICECCS)*. IEEE Computer Society, 2005, pp. 303–312.

[34] Y. Bontemps, P. Heymans, P.-Y. Schobbens, and J.-C. Trigaux, "Semantics of Feature Diagrams," in *Proc. Int'l Workshop on Software Variability Management for Product Derivation - Towards Tool Support (SVMPD)*, 2004.

[35] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin, "Generating Product-Lines of Product-Families," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE Computer Society, 2002, pp. 81–92.

[36] J. White, B. Dougherty, and D. C. Schmidt, "Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening," *J. of Systems and Software (JSS)*, vol. 82, no. 8, pp. 1268–1284, 2009.

[37] M. Mendonca, M. Branco, and D. Cowan, "S.P.L.O.T.: Software Product Lines Online Tools," in *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2009, pp. 761–762.

[38] R. Zippel, "LinuxKernelConf Documentation," Website, available online at http://www.xs4all.nl/~zippel/lc/lkc-language.txt; visited on June 18th, 2010.

[39] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability Modeling in the Real: a Perspective from the Operating Systems Domain," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, 2010, pp. 73–82.

[40] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow, "Extending Feature Diagrams with UML Multiplicities," in *Proc. World Conf. Integrated Design and Process Technology (IDPT)*, 2002.

[41] R. Michel, A. Classen, A. Hubaux, and Q. Boucher, "A Formal Semantics for Feature Cardinalities in Feature Diagrams," in *Proc. Workshop Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 2011, pp. 82–89.

[42] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés, "Using Java CSP Solvers in the Automated Analyses of Feature Models," in *Proc. Generative and Transformational Techniques in Software Engineering*, ser. LNCS, vol. 4143. Springer, 2006, pp. 399–408.

[43] M. Janota and J. Kiniry, "Reasoning about Feature Models in Higher-Order Logic," in *Proc. Int'l Software Product Line Conference (SPLC)*. IEEE Computer Society, 2007, pp. 13–22.

[44] K. Czarnecki and A. Wasowski, "Feature Diagrams and Logics: There and Back Again," in *Proc. Int'l Software Product Line Conference (SPLC)*. IEEE Computer Society, 2007, pp. 23–34.

[45] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse Engineering Feature Models," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2011, pp. 461–470.