Formal Approach to Integrating Feature and Architecture Models

Mikoláš Janota[1] and Goetz Botterweck[2]

[1] School of Computer Science and Informatics,
Lero, University College Dublin, Dublin, Ireland
mikolas.janota@ucd.ie
[2] Lero, University of Limerick, Limerick, Ireland
goetz.botterweck@lero.ie

Abstract. If we model a family of software applications with a feature model and an architecture model, we are describing the same subject from different perspectives. Hence, we are running the risk of inconsistencies. For instance, the feature model might allow feature configurations that are not realizable by the architecture.
In this paper we tackle this problem by providing a formalization of dependencies between features and components. Further, we demonstrate that this formalization offers a better understanding of the modeled concepts. Moreover, we propose automated techniques that derive additional information and provide feedback to the user. Finally, we discuss how some of these techniques can be implemented.

1 Introduction

Many companies providing software-intensive systems have to deal with the challenge to fulfill the expectation of each individual customer and, at the same time, to perform the necessary engineering processes in an efficient manner.

One way to approach this challenge is by focusing on building a whole set of similar systems in a systematic way. This phenomenon shaped a field known as *software product lines (SPL)* [6]. A software product line is "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment [..] and that are developed from a common set of core assets in a prescribed way" [21]. SPL approaches use miscellaneous models to describe the various aspects of a product-line. This includes feature models [13,8] or architecture models containing the components that implement the features. We will now use very simple examples of such models to illustrate the motivation for our research presented in this paper.

Imagine a product-line of mobile phones that has only two features which distinguish the various products: the capability to play MP3 files, and a built-in camera. Fig. 1 shows the corresponding models. In SPL engineering such models are created by the *domain engineer* in order to describe the scope of the whole product-line. Further, these models are given to the *application engineer* to be used during the configuration and derivation of a concrete product.

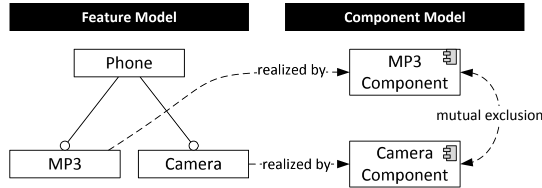

Fig. 1. Feature model and component model of a mobile phone product-line

The feature model in Fig. 1 does not capture any dependencies between the features `MP3` and `Camera`. Nevertheless, the feature `MP3` is realized by `MP3Component`, the feature `Camera` is realized by `CameraComponent`, and there is a mutual exclusion between the two components. This means that the feature model permits a combination of features that is not realizable with respect to the component model. In other words, there is an *implicit* dependency (exclusion) between the two features.

This poses a problem, as during the feature configuration the application engineer often deals solely with the feature model since it is simply too time-consuming to walk through subsequent implementation steps (to check each possible feature configuration for its realizability). In this small example the chain of dependencies is easy to spot. In real SPL projects with their complex models and large number of dependencies these implicit dependencies pose a big problem. Hence, it is desirable to enhance the feature model in Fig. 1 with the mutual exclusion between the features `MP3` and `Camera`.

In this article we are focusing on how to find such implicit (missing) dependencies. The feature model enhanced with the missing dependencies will provide a better guidance to the application engineering during the configuration process as the additional dependencies disallow void configurations. We are offering a means of finding these implicit dependencies by providing a semantics for the involved models and analyses of this semantics.

The remainder of this text is structured as follows. First we provide a background on basic concepts (Sect. 2). We then give an overview of our approach (Sect. 3) and explain it for the general case (Sect. 4). We discuss how the relation "realized by" can be interpreted (Sect. 5) and specialize the general case for propositional logic (Sect. 6). This includes the automatic calculation of the defined properties, which has been implemented in a prototype (Sect. 6.1). We conclude with an overview of related work (Sect. 7), a short summary, and an outlook on potential topics for future work (Sect. 8).

2 Background

Kang et al. define a feature as "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" [13]. In the literature there are numerous suggestions for feature modeling languages, e.g., in [13,7].

Defining the semantics of a modeling language in a mathematical form is preferred, because it rules out ambiguity. For instance, for the feature model in Fig. 1 we demand that a feature cannot be selected without its parent, and, that the root is always selected. This is expressed by the following formulas.

root must be selected:	*children require parent:*
Phone	MP3 $\Rightarrow$ Phone
	Camera $\Rightarrow$ Phone

Other primitives of feature modeling languages are mapped to their formal representation in a similar fashion. For instance, the missing exclusion would be expressed as $\neg(\text{MP3} \wedge \text{Camera})$.

Such mappings from models to formal descriptions are used as a foundation for (automated) reasoning. This area has been addressed by a number of researchers [16,2,19], including our own work on formalizing feature models in higher-order logic [12].

Feature models are not the only way to describe the capabilities of a product line. For instance, a *domain architecture* takes a more solution-oriented perspective and describes potential implementation structures. This includes variable elements that can be configured and, hence, enables the derivation of different *product-specific architectures* from the domain architecture. In this paper we will focus on the components contained in such an architecture and abstract from other details. We will use a *component model* to capture constraints for the whole product line and *component configurations* to describe the components selected for one particular product.

Analogously to the feature model, the component model is interpreted by mapping it to a formal representation. For instance, the mutual exclusion in the example component model in Fig. 1 is interpreted as

$$\neg(\text{MP3Component} \wedge \text{CameraComponent})$$

All these mappings depend on the interpretation of the particular feature or component modeling language and the type of logic used as a formal representation. It should be mentioned that the approach presented in this paper is designed to be independent of particular meta-models and independent of a specific forms of logic (e.g., propositional logic or first order logic).

3 Overview of Our Approach

We will now start to conceptualize our approach by defining four interrelated models. We identified two major aspects that we want to concentrate on, features and components. These can be modeled on two levels, resembling the distinction between product line and products.

A *feature model* (see ❶ in Fig. 2) describes the capabilities of the product line and defines the constraints for potential products. A *feature configuration* ❷ describes the features selected for one particular product. It no longer contains

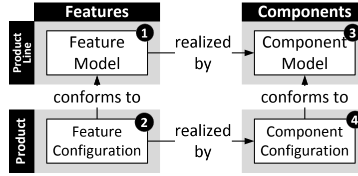

Fig. 2. Two aspects (features, components) on two levels (model, configuration)

variability since all configuration decisions have been made. It can be checked for conformity with the feature model.

Analogously a *component model* ❸ describes the elements used in the implementation of the products. A *component configuration* ❹ describes which components have been selected for a particular product.

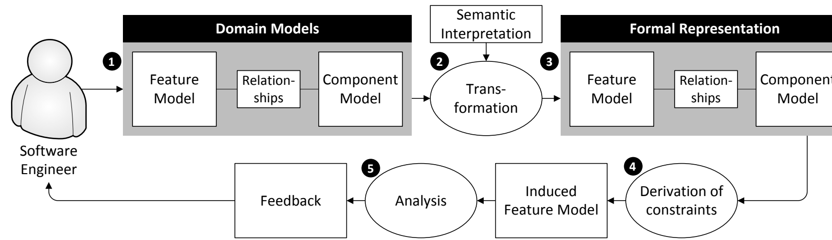

Fig. 3. Overview of our approach

We proceed by describing an overview of our approach (see Fig. 3): The domain engineer starts by creating models of features and the related components ❶. By applying a semantic interpretation ❷ these models are transformed into formal representations ❸.

This formal representation is then used to derive ❹ additional information which then is further analyzed ❺. The resulting information is used to provide feedback to the domain engineer.

In the end, this additional information should be used to enhance the feature model to provide guidance to the application engineer during the configuration process. More specifically, the contributions of our article are as follows: (1) We provide a formalization that integrates features, components, and the dependencies between them. This formalization is independent of particular feature or architecture modeling languages. (2) We specialize the formalization for models that are expressible in propositional logic. (3) This enables the implementation of automatic reasoning techniques to derive additional constraints from the models. (4) Consequently, we are able to cut down the configuration space and provide the desired guidance to the software engineer.

4 Feature-Component Models: General Case

Feature models appear in a variety of forms. They can have attributes, cardinalities and various types of constraints. For the following discussion, we will abstract from these different representations and instead directly operate on the actual semantics. Hence, we will treat a model as an oracle which, given a particular configuration, answers 'yes' or 'no' to indicate whether that configuration conforms to the model or not.

4.1 Definitions of the Basic Concepts

When modeling a particular product line, we assume that we are given a *set of feature configurations*, denoted as $\mathbb{F}$, and a *set of component configurations*, denoted as $\mathbb{C}$.

In this text, we will represent constraints on configurations as sets. A given configuration satisfies a given constraint if and only if this configuration is an element of the set representing that constraint. Taking this approach, the following definitions establish the building blocks for the rest of this article. We begin by defining entities that capture constraints on features and components.

Definition 1 (feature and component models).

1. *a* feature model $\mathcal{M}_\mathrm{f}$ *is a set of feature configurations:* $\mathcal{M}_\mathrm{f} \subseteq \mathbb{F}$
2. *a feature configuration* $\mathbf{f} \in \mathbb{F}$ conforms *to the feature model* $\mathcal{M}_\mathrm{f}$ *if and only if* $\mathbf{f} \in \mathcal{M}_\mathrm{f}$
3. *a* component model $\mathcal{M}_\mathrm{c}$ *is a set of component configurations:* $\mathcal{M}_\mathrm{c} \subseteq \mathbb{C}$
4. *a component configuration* $\mathbf{c} \in \mathbb{C}$ conforms *to the component model* $\mathcal{M}_\mathrm{c}$ *if and only if* $\mathbf{c} \in \mathcal{M}_\mathrm{c}$

Building on the preceding definition, we define concepts for capturing constraints on features and components together.

Definition 2 (feature-component model).

1. *a* feature-component configuration $\langle \mathbf{f}, \mathbf{c} \rangle$ *is a pair consisting of a feature configuration and a component configuration:* $\langle \mathbf{f}, \mathbf{c} \rangle \in \mathbb{F} \times \mathbb{C}$
2. *a* feature-component model $\mathcal{M}_\mathrm{fc}$ *is a set of feature-component configurations:* $\mathcal{M}_\mathrm{fc} \subseteq \mathbb{F} \times \mathbb{C}$
3. *a feature-component configuration* $\langle \mathbf{f}, \mathbf{c} \rangle \in \mathbb{F} \times \mathbb{C}$ conforms to *the feature-component model* $\mathcal{M}_\mathrm{fc}$ *if and only if* $\langle \mathbf{f}, \mathbf{c} \rangle \in \mathcal{M}_\mathrm{fc}$

The following two examples illustrate the concepts introduced by Defs. 1 and 2.

Example 1. Let us consider a case with two features, f_1 and f_2, and two components, c_1 and c_2. We will represent a configuration as a set of the features or components that are selected — a feature or component not in the set is unselected. Hence, feature configurations will correspond to the subsets of $\{f_1, f_2\}$; similarly, the component configurations will correspond to the subsets of $\{c_1, c_2\}$.

To express this in a mathematical notation, we utilize the concept of a powerset, denoted $\mathcal{P}(\cdot)$, as follows: $\mathbb{F} \equiv \mathcal{P}(\{f_1, f_2\})$ and $\mathbb{C} \equiv \mathcal{P}(\{c_1, c_2\})$. The feature model $\mathcal{M}_\text{f}$ requires that at least one feature is selected and component model $\mathcal{M}_\text{c}$ states that whenever c_1 is selected, c_2 must be selected, expressed as follows.

$$\mathcal{M}_\text{f} \equiv \mathbb{F} \setminus \{\emptyset\} = \{\{f_1\}, \{f_2\}, \{f_1, f_2\}\}$$
$$\mathcal{M}_\text{c} \equiv \{\mathbf{c} \in \mathbb{C} \mid c_1 \in \mathbf{c} \Rightarrow c_2 \in \mathbf{c}\} = \{\emptyset, \{c_2\}, \{c_1, c_2\}\}$$

We introduce an auxiliary relation $\mathcal{R}$ to impose additional constraints on the combinations of feature and component configurations:

$$\mathbf{f}\,\mathcal{R}\,\mathbf{c} \Leftrightarrow ((f_1 \in \mathbf{f} \Rightarrow c_1 \in \mathbf{c}) \wedge (f_2 \in \mathbf{f} \Rightarrow c_2 \in \mathbf{c}))$$

(Note that the relation $\mathcal{R}$ states that f_1 requires c_1 and f_2 requires c_2.)

To obtain the overall feature-component model $\mathcal{M}_\text{fc}$ we combine all these constraints together: $\mathcal{M}_\text{fc} \equiv (\mathcal{M}_\text{f} \times \mathcal{M}_\text{c}) \cap \mathcal{R}$.

In plain language, a feature-component configuration conforms to $\mathcal{M}_\text{fc}$ if and only if its feature part conforms to $\mathcal{M}_\text{f}$, its component part conforms to $\mathcal{M}_\text{c}$, and the whole pair belongs to the relation $\mathcal{R}$.

For this feature-component model, $\langle\emptyset,\ \{c_1\}\rangle$ and $\langle\{f_2\},\ \{c_1\}\rangle$ are examples of non-conforming configurations, whereas $\langle\{f_1\},\ \{c_1, c_2\}\rangle$ is a conforming configuration.

Example 2. Let the set of feature configurations $\mathbb{F}$ where each configuration is a pair consisting of the set of selected features and a natural number representing an attribute of the feature f_1: $\mathbb{F} \equiv \mathcal{P}(\{f_1, f_2\}) \times \mathbb{N}$. Let the feature model disallow selecting the feature f_2 when f_1 is selected with the value less than 100:

$$\mathcal{M}_\text{f} \equiv \{\langle\text{sel, attr}_1\rangle \in \mathbb{F} \mid (f_1 \in \text{sel} \wedge \text{attr}_1 < 100) \Rightarrow f_2 \notin \text{sel}\}$$

Let $\mathbb{C} \equiv \mathcal{P}(\{c_1, c_2\})$ and $\mathcal{M}_\text{c} \equiv \mathbb{C}$, i.e., $\mathcal{M}_\text{c}$ is imposing no restrictions on the component configurations. To express how features relate to components, we define the following three relations:

$$(\langle\text{sel, attr}_1\rangle\,\mathcal{R}_1\,\mathbf{c}) \Leftrightarrow (f_1 \in \text{sel} \wedge \text{attr}_1 < 500 \Rightarrow c_1 \in \mathbf{c})$$
$$(\langle\text{sel, attr}_1\rangle\,\mathcal{R}_2\,\mathbf{c}) \Leftrightarrow (f_1 \in \text{sel} \wedge \text{attr}_1 \not< 500 \Rightarrow c_1 \in \mathbf{c} \wedge c_2 \in \mathbf{c})$$
$$(\langle\text{sel, attr}_1\rangle\,\mathcal{R}_3\,\mathbf{c}) \Leftrightarrow (f_2 \in \text{sel} \Rightarrow c_2 \in \mathbf{c})$$

The relation $\mathcal{R}_1$ expresses that if the feature f_1 is selected and the value of its attribute is less than 500, then the component c_1 is required. The relation $\mathcal{R}_2$, however, requires the component c_2 must be selected on top of c_1 once the attribute's value is not less than 500. The relation $\mathcal{R}_3$ records that the feature f_2 requires the component c_2. Let relation $\mathcal{R}$ be the intersection of these relations $\mathcal{R} \equiv \mathcal{R}_1 \cap \mathcal{R}_2 \cap \mathcal{R}_3$, then all the constraints combined are $\mathcal{M}_\text{fc} \equiv (\mathcal{M}_\text{f} \times \mathcal{M}_\text{c}) \cap \mathcal{R}$.

4.2 From Overall Constraints to Feature Models

In the introduction we have promised that we will investigate how overall constraints, constraints on a composite feature-component model, are projected back onto the features. The following definition formalizes the meaning of this projection.

Definition 3 (induced feature model). *For a feature-component model $\mathcal{M}_{\mathrm{fc}}$, the* induced feature model $\mathcal{I}_{\mathcal{M}_{\mathrm{fc}}} \subseteq \mathbb{F}$ *is a set of feature configurations for which there exists a component configuration such that together they conform to the feature-component model:*

$$\mathcal{I}_{\mathcal{M}_{\mathrm{fc}}} \equiv \{\mathbf{f} \in \mathbb{F} \mid (\exists \mathbf{c} \in \mathbb{C}) \, \langle \mathbf{f}, \, \mathbf{c} \rangle \in \mathcal{M}_{\mathrm{fc}}\}$$

Intuitively, for any feature configuration conforming to the induced feature model, we are *guaranteed* that there is an implementation of this feature configuration (with respect to the pertaining feature-component model). On the other hand, if the induced feature model is not equal to the feature model given by the user, it means that the feature model permits a feature configuration without implementation and thus should be improved. This is illustrated by the following example.

Example 3. Let $\mathbb{F} \equiv \mathcal{P}(\{f_1, f_2\})$, $\mathbb{C} \equiv \mathcal{P}(\{c_1, c_2\})$, and the feature-component model $\mathcal{M}_{\mathrm{fc}}$ defined as follows.

$$\begin{array}{l} \mathcal{M}_{\mathrm{f}} \equiv \mathbb{F} \quad \mathcal{M}_{\mathrm{c}} \equiv \mathbb{C} \smallsetminus \{c_1, c_2\} \\ \mathbf{f} \, \mathcal{R} \, \mathbf{c} \Leftrightarrow ((f_1 \in \mathbf{f} \Rightarrow c_1 \in \mathbf{c}) \wedge (f_2 \in \mathbf{f} \Rightarrow c_2 \in \mathbf{c})) \\ \mathcal{M}_{\mathrm{fc}} \equiv (\mathcal{M}_{\mathrm{f}} \times \mathcal{M}_{\mathrm{c}}) \cap \mathcal{R} \end{array}$$

We see that f_1 and f_2 require c_1 and c_2, respectively, and that c_1 and c_2 are mutually excluded. Therefore, there is no implementation for the feature configuration $\{f_1, f_2\}$, even though it is permitted by the feature model $\mathcal{M}_{\mathrm{f}}$. In other words, the induced feature model for $\mathcal{M}_{\mathrm{fc}}$ is $(\mathbb{F} \smallsetminus \{f_1, f_2\})$.

4.3 Configurations with Preference

In this section we consider that the software engineer *prefers* some configurations over other configurations. To motivate further discussion let us look at some of the configurations for Example 1. The configuration $\langle \{f_1\}, \, \{c_1, c_2\} \rangle$ conforms to $\mathcal{M}_{\mathrm{fc}}$, but we are not getting the best out of the included components as the feature f_2 could be added with no further consequences (no additional component is required). However, it is not possible to remove any of the components while keeping the feature f_1. Dually, the configuration $\langle \{f_2\}, \, \{c_1, c_2\} \rangle$ uses more components than necessary, i.e., c_1 can be removed while keeping the same features. So we can say that we *prefer* configurations with more features and with less components.

To record this formally, we assume that we are given two partial orderings, one on feature configurations, denoted as $\sqsubseteq_{\mathrm{f}}$, and one on component configurations, denoted as $\sqsubseteq_{\mathrm{c}}$. Intuitively, $\sqsubseteq_{\mathrm{f}}$ corresponds to the increase of capabilities expressed by the features; whereas $\sqsubseteq_{\mathrm{c}}$ corresponds to the weight, or price, ordering on component configurations and we will prefer cheaper solutions to the expensive ones. To reflect the discussion above, for Example 1 we define the orderings as $\sqsubseteq_{\mathrm{f}} \equiv \subseteq$ and $\sqsubseteq_{\mathrm{c}} \equiv \subseteq$. For Example 2, we can, for instance, define the orderings in the same fashion by ignoring the feature attribute as follows.

$$\left(\langle \mathrm{sel},\, \mathrm{attr}_1 \rangle \sqsubseteq_{\mathrm{f}} \left\langle \mathrm{sel}',\, {\mathrm{attr}_1}' \right\rangle\right) \Leftrightarrow \left(\mathrm{sel} \subseteq \mathrm{sel}'\right) \qquad (\mathbf{c} \sqsubseteq_{\mathrm{c}} \mathbf{c}') \Leftrightarrow (\mathbf{c} \subseteq \mathbf{c}')$$

The following definition utilizes these orderings to characterize feature-component configurations.

Definition 4 (strong conformity). *For a feature-component model $\mathcal{M}_{\mathrm{fc}}$. A configuration $\langle \mathbf{f},\, \mathbf{c} \rangle$* strongly conforms *to $\mathcal{M}_{\mathrm{fc}}$ if and only if $\langle \mathbf{f},\, \mathbf{c} \rangle \in \mathcal{M}_{\mathrm{fc}}$ and the following holds.*

$$(\forall \langle \mathbf{f}',\, \mathbf{c}' \rangle \in \mathcal{M}_{\mathrm{fc}})((\mathbf{f} \sqsubseteq_{\mathrm{f}} \mathbf{f}' \wedge \mathbf{c}' \sqsubseteq_{\mathrm{c}} \mathbf{c}) \Rightarrow (\mathbf{f} = \mathbf{f}' \wedge \mathbf{c} = \mathbf{c}'))$$

Intuitively, we cannot add features to a strong conforming configuration without adding components, or, remove components without reducing the features. In Example 1, the configuration $\langle \{f_2\},\, \{c_2\} \rangle$ strongly conforms to $\mathcal{M}_{\mathrm{fc}}$. Whereas, the configuration $\langle \{f_1\},\, \{c_1, c_2\} \rangle$ does not strongly conform to $\mathcal{M}_{\mathrm{fc}}$ because we can add the feature f_2 (the required component c_2 is already in place). Nevertheless, the cost of the configuration cannot be improved while keeping the feature f_1 as this feature requires both components.

5 Semantics of "realized by"

As we have seen in the preceding examples, it is natural to express feature-component models in the form $(\mathcal{M}_{\mathrm{f}} \times \mathcal{M}_{\mathrm{c}}) \cap \mathcal{R}$. The user specifies the feature model $\mathcal{M}_{\mathrm{f}}$ in a feature modeling language and $\mathcal{M}_{\mathrm{c}}$ in an architecture modeling language that supports variability. Both types of languages have been widely studied from various angles and tool support exists. Little work, however, has been done to study the glue between the two representations, the relation $\mathcal{R}$ in our formalism.

We would like to enable the domain engineer to express herself at a higher level of abstraction — using concepts as "The feature f is realized by the component c." First, we need to define a language concept for expressing such facts and second, provide semantics for it in a mathematical form.

It might seem, at first glance, that it is sufficient to have a single mapping from each feature to the component that realizes that feature. In practice, however, we need to cope with more complex scenarios. For instance, when a feature can be implemented by different components, or, when a feature requires multiple components. Dually, a combination of features as a whole might impose different requirements in contrast to the combination of the requirements imposed by each of them. Hence, we introduce a language construct that maps sets of feature configurations to sets of component configurations.

Definition 5. *Let $\mathcal{S}_{\mathrm{f}} \subseteq \mathbb{F}$ and $\mathcal{S}_{\mathrm{c}} \subseteq \mathbb{C}$, then* $\text{realized-by}(\mathcal{S}_{\mathrm{f}}, \mathcal{S}_{\mathrm{c}})$ *is a* realized-by expression. *If a realized-by expression is in the form* $\text{realized-by}(\{\mathbf{f} \in \mathbb{F} \mid f_{\mathrm{r}} \in \mathbf{f}\}, \mathcal{S}_{\mathrm{c}})$ *then we say that the feature f_{r} is* realized by $\mathcal{S}_{\mathrm{c}}$.

Example 4. Let $\mathbb{F} \equiv \mathcal{P}(\{f_1, f_2\})$ and $\mathbb{C} \equiv \mathcal{P}(\{c_1, c_2, c_3\})$

$$\begin{array}{l}\text{realized-by}(\{\mathbf{f} \in \mathbb{F} \mid f_2 \in \mathbf{f}\}, \{\mathbf{c} \in \mathbb{C} \mid c_1 \in \mathbf{c} \vee c_2 \in \mathbf{c}\}) \\ \text{realized-by}(\{\mathbf{f} \in \mathbb{F} \mid \{f_1, f_2\} \subseteq \mathbf{f}\}, \{\mathbf{c} \in \mathbb{C} \mid \{c_1, c_2\} \subseteq \mathbf{c}\})\end{array}$$

Intuitively, each realized-by expression imposes a restriction on the feature-component configurations. The first one in Example 4 specifies that either of c_1 or c_2 is an implementation of f_2; the second expression specifies that the combination c_1 and c_2 is an implementation of the combination f_1 and f_2.

Before we proceed with the semantics of the realized-by expressions, we will put them in the context of feature and component models.

Definition 6. *A* product line model *is a triple* $\langle \mathcal{M}_\mathrm{f}, \mathcal{M}_\mathrm{c}, Q \rangle$*, where* $\mathcal{M}_\mathrm{f}$ *is a feature model,* $\mathcal{M}_\mathrm{c}$ *is a component model, and* Q *is a set of realized-by expressions.*

To formally analyze this model, we have to define our interpretation of it (see ❷ in Fig. 2). For this, we present three alternatives:

1. Interpretation with $\Rightarrow$
2. Interpretation with $\Leftrightarrow$
3. Interpretation with $\Rightarrow$ and strong-conformity (see Def. 4)

Definition 7. Semantics *of product line models is a function that maps a software product line model to a feature-component model. We will use the notation* $[\![\cdot]\!]$ *for a function of the type* $\langle \mathcal{M}_\mathrm{f}, \mathcal{M}_\mathrm{c}, Q \rangle \rightarrow \mathcal{P}(\mathbb{F} \times \mathbb{C})$.

1. *Interpretation with* $\Rightarrow$*. Let the relation* $\mathcal{R} \subseteq \mathbb{F} \times \mathbb{C}$ *be defined as follows*

$$\mathbf{f}\,\mathcal{R}\,\mathbf{c} \Leftrightarrow \bigwedge_{\text{realized-by}(\mathcal{S}_\mathrm{f}, \mathcal{S}_\mathrm{c}) \in Q} \mathbf{f} \in \mathcal{S}_\mathrm{f} \Rightarrow \mathbf{c} \in \mathcal{S}_\mathrm{c}$$

Then $[\![\langle \mathcal{M}_\mathrm{f}, \mathcal{M}_\mathrm{c}, Q \rangle]\!]_1 \equiv (\mathcal{M}_\mathrm{f} \times \mathcal{M}_\mathrm{c}) \cap \mathcal{R}$*. In other words, the inclusion of features implies the inclusion of related components — but not necessarily the other way around.*

2. *Interpretation with* $\Leftrightarrow$*. Let the relation* $\mathcal{R} \subseteq \mathbb{F} \times \mathbb{C}$ *be defined as follows*

$$\mathbf{f}\,\mathcal{R}\,\mathbf{c} \Leftrightarrow \bigwedge_{\text{realized-by}(\mathcal{S}_\mathrm{f}, \mathcal{S}_\mathrm{c}) \in Q} \mathbf{f} \in \mathcal{S}_\mathrm{f} \Leftrightarrow \mathbf{c} \in \mathcal{S}_\mathrm{c}$$

Then $[\![\langle \mathcal{M}_\mathrm{f}, \mathcal{M}_\mathrm{c}, Q \rangle]\!]_2 \equiv (\mathcal{M}_\mathrm{f} \times \mathcal{M}_\mathrm{c}) \cap \mathcal{R}$*. In other words, the inclusion of features implies the inclusion of related components — and vice versa.*

3. *Interpretation with* $\Rightarrow$ *and strong conformity. Given the orderings* $\sqsubseteq_\mathrm{f}$ *and* $\sqsubseteq_\mathrm{c}$*. Let* $\mathcal{M}_\mathrm{fc} = [\![\langle \mathcal{M}_\mathrm{f}, \mathcal{M}_\mathrm{c}, Q \rangle]\!]_1$ *then*

$$[\![\langle \mathcal{M}_\mathrm{f}, \mathcal{M}_\mathrm{c}, Q \rangle]\!]_3 \equiv \{\langle \mathbf{f}, \mathbf{c} \rangle \in \mathbb{F} \times \mathbb{C} \mid \langle \mathbf{f}, \mathbf{c} \rangle \textit{ strongly conforms to } \mathcal{M}_\mathrm{fc} \textit{ w.r.t. } \sqsubseteq_\mathrm{f} \textit{ and } \sqsubseteq_\mathrm{c} \}$$

In other words, we use the first interpretation and in addition require strong-conformity. Hence, we reduce the feature-component models to those configurations that cannot be improved in their capability or cost.

The following examples will illustrate the different semantics on two product line models, $\mathrm{PLM_a}$ and $\mathrm{PLM_b}$, with $\mathbb{F} \equiv \mathcal{P}(\{f_1, f_2\})$, $\mathbb{C} \equiv \mathcal{P}(\{c_1, c_2\})$, and the orderings $\sqsubseteq_\mathrm{f} \equiv \subseteq$ and $\sqsubseteq_\mathrm{c} \equiv \subseteq$.

Example 5. Let $\mathrm{PLM_a} \equiv \langle \mathcal{M}_\mathrm{f}, \mathcal{M}_\mathrm{c}, Q \rangle$ be a product line model where:

$$\begin{aligned}
&\mathcal{M}_\mathrm{f} \equiv \mathbb{F} \qquad \mathcal{M}_\mathrm{c} \equiv \{\mathbf{c} \in \mathbb{C} \mid c_1 \in \mathbf{c} \Rightarrow c_2 \in \mathbf{c}\} \\
&Q \equiv \{\, \text{realized-by}(\{\mathbf{f} \in \mathbb{F} \mid f_1 \in \mathbf{f}\}, \{\mathbf{c} \in \mathbb{C} \mid c_1 \in \mathbf{c}\}), \\
&\qquad \text{realized-by}(\{\mathbf{f} \in \mathbb{F} \mid f_2 \in \mathbf{f}\}, \{\mathbf{c} \in \mathbb{C} \mid c_2 \in \mathbf{c}\}) \,\}
\end{aligned}$$

According to Def. 7, the semantics $[\![\cdot]\!]_1$ and $[\![\cdot]\!]_2$ of $\mathrm{PLM_a}$ correspond to the following feature-component models.

$$\begin{aligned}
&\langle \mathbf{f}, \mathbf{c} \rangle \in [\![\mathrm{PLM_a}]\!]_1 \Leftrightarrow && \langle \mathbf{f}, \mathbf{c} \rangle \in [\![\mathrm{PLM_a}]\!]_2 \Leftrightarrow \\
&\quad f_1 \in \mathbf{f} \Rightarrow c_1 \in \mathbf{c} \wedge && \quad f_1 \in \mathbf{f} \Leftrightarrow c_1 \in \mathbf{c} \wedge \\
&\quad f_2 \in \mathbf{f} \Rightarrow c_2 \in \mathbf{c} \wedge && \quad f_2 \in \mathbf{f} \Leftrightarrow c_2 \in \mathbf{c} \wedge \\
&\quad c_1 \in \mathbf{c} \Rightarrow c_2 \in \mathbf{c} && \quad c_1 \in \mathbf{c} \Rightarrow c_2 \in \mathbf{c}
\end{aligned}$$

To obtain the semantics $[\![\mathrm{PLM_a}]\!]_3$ we compute the strongly conforming configurations of $[\![\mathrm{PLM_a}]\!]_1$, which are the following.

$$[\![\mathrm{PLM_a}]\!]_3 \equiv \{\langle \emptyset, \emptyset \rangle, \langle \{f_2\}, \{c_2\} \rangle, \langle \{f_1, f_2\}, \{c_1, c_2\} \rangle\}$$

Interestingly, the feature component models $[\![\mathrm{PLM_a}]\!]_2$ and $[\![\mathrm{PLM_a}]\!]_3$ are equal. Intuitively, the reason for this is that once c_2 is included in a configuration, there is nothing that prevents f_2 from being included.

If we look at the resulting models from the perspective of induced feature models (see Def. 3), we see that the induced feature model for $[\![\mathrm{PLM_a}]\!]_1$ is imposing no restrictions on the features. Here the user might feel that there is some information lost, since originally there was a dependency between the realizing components but now there is no dependency between the features themselves.

For $[\![\mathrm{PLM_a}]\!]_2$ and $[\![\mathrm{PLM_a}]\!]_3$ the induced feature model is $\{\mathbf{f} \in \mathbb{F} \mid f_1 \in \mathbf{f} \Rightarrow f_2 \in \mathbf{f}\}$, due to the bi-implication between the inclusion of a feature and the inclusion of the implementing component.

Hence, for $\mathrm{PLM_a}$ the semantics $[\![\cdot]\!]_2$ and $[\![\cdot]\!]_3$ return the same result and they project dependencies between components to dependencies between features. Whereas the semantics $[\![\cdot]\!]_1$ does not project dependencies between the components onto the features.

The next example, $\mathrm{PLM_b}$, strengthens the constraints of $\mathrm{PLM_a}$ by adding an exclusions between the two features.

Example 6. Let $\mathrm{PLM_b} \equiv \langle \mathcal{M}_\mathrm{f}, \mathcal{M}_\mathrm{c}, Q \rangle$ be a product line model such that:

$$\begin{aligned}
\mathcal{M}_\mathrm{f} &\equiv \{\mathbf{f} \in \mathcal{P}(\{f_1, f_2\}) \mid \neg(f_1 \in \mathbf{f} \wedge f_2 \in \mathbf{f})\} \\
\mathcal{M}_\mathrm{c} &\equiv \{\mathbf{c} \in \mathcal{P}(\{c_1, c_2\}) \mid c_1 \in \mathbf{c} \Rightarrow c_2 \in \mathbf{c}\} \\
Q &\equiv \{\text{realized-by}(\{\mathbf{f} \in \mathbb{F} \mid f_1 \in \mathbf{f}\}, \{\mathbf{c} \in \mathbb{C} \mid c_1 \in \mathbf{c}\}), \\
&\qquad \text{realized-by}(\{\mathbf{f} \in \mathbb{F} \mid f_2 \in \mathbf{f}\}, \{\mathbf{c} \in \mathbb{C} \mid c_2 \in \mathbf{c}\})\}
\end{aligned}$$

The three semantics yield the following feature-component models (enumerated as conforming feature-component configurations).

$$[\![\mathrm{PLM_b}]\!]_1 \equiv \{\langle\emptyset,\,\emptyset\rangle\,,\langle\{f_1\},\,\{c_1,c_2\}\rangle\,,\langle\{f_2\},\,\{c_2\}\rangle\,,\langle\{f_2\},\,\{c_1,c_2\}\rangle\}$$
$$[\![\mathrm{PLM_b}]\!]_3 \equiv \{\langle\emptyset,\,\emptyset\rangle\,,\langle\{f_1\},\,\{c_1,c_2\}\rangle\,,\langle\{f_2\},\,\{c_2\}\rangle\}$$
$$[\![\mathrm{PLM_b}]\!]_2 \equiv \{\langle\emptyset,\,\emptyset\rangle\,,\langle\{f_2\},\,\{c_2\}\rangle\}$$

Now, let us compare these sets. The feature-component model given by $[\![\cdot]\!]_1$ is formed by exactly the combinations where the implementation provides sufficient functionality for the selected features. The semantics $[\![\cdot]\!]_3$ additionally 'filters out' the configuration $\langle\{f_2\},\,\{c_1,c_2\}\rangle$ as c_1 is not needed for f_2.

The semantics $[\![\cdot]\!]_2$, however, yields a somewhat surprising feature-component model. Due to the bi-implication between features and realizing components, f_1 is not selectable since if we are to select f_1, we need to select c_1 which requires c_2 but f_2 cannot be selected due to the restriction imposed by the feature model. A feature that does not appear in any product (is not selectable), is called a *dead feature* and it is most likely an undesired property of the feature-component model.

Hence, for $\mathrm{PLM_b}$, each of the semantics yields a different feature-component model. Note, however, that $[\![\mathrm{PLM_b}]\!]_1$ and $[\![\mathrm{PLM_b}]\!]_3$ have the same induced feature model. The semantics $[\![\cdot]\!]_2$ appears to be inappropriate for $\mathrm{PLM_b}$ as the feature f_2 is dead under this semantics.
To conclude this section, a summary of the observations follows.

1. translate via implication, the relation between features and components is somewhat "loose": the features require their implementations but there is no dependency in the other direction,
2. translate via bi-implication, the resulting model requires for the features to be "on" whenever they are implemented; this dependency might bee too strong in some cases,
3. translate via implication and strong-conformity, in that case a feature is required to be "switched on" whenever it is *possible* to switch it on.

6 Feature-Component Models with Propositional Logic

In this section we specialize the concepts introduced so far for propositional logic. We focus on features and components that can just be selected or deselected (in contrast to having attributes) and hence, can be expressed as boolean variables.

Before we proceed, let us recall some basic concepts from propositional logic. Let F be a boolean formula on the set of variables V, $m \subseteq V$, and let F' be obtained from F by replacing every variable $v \in m$ by *true* and every variable $v' \notin m$ by *false*, then m is a *model* of F if and only if F' evaluates to *true*. For the evaluation we assume the standard semantics of boolean connectives, such as *true* $\wedge$ *false* evaluates to *false*.

For the following we assume $\mathcal{F}$ to be a finite set of features and $\mathcal{C}$ a finite set of components such that $\mathcal{F} \cap \mathcal{C} = \emptyset$. The domain of feature configurations

then becomes $\mathbb{F} \equiv \mathcal{P}(\mathcal{F})$ and the domain of component configurations becomes $\mathbb{C} \equiv \mathcal{P}(\mathcal{C})$. We will say that a feature or component is *selected* by a configuration if and only if it is in that configuration. We will use the subset relation to define the orderings on configurations, i.e., $\sqsubseteq_{\rm f} \equiv \subseteq$ and $\sqsubseteq_{\rm c} \equiv \subseteq$.

Feature and component models will be represented as boolean formulas on the variables $\mathcal{F}$ and $\mathcal{C}$, respectively; the feature-component models will be represented as formulas on the variables $\mathcal{F} \cup \mathcal{C}$ (recall that $\mathcal{F}$ and $\mathcal{C}$ are disjoint). The conforming configurations will correspond to models of the formulas. The correspondence between the general form and the boolean representation is illustrated by the following example.

Example 7. For the model in Example 1 we have $\mathcal{F} \equiv \{f_1, f_2\}$, $\mathcal{C} \equiv \{c_1, c_2\}$. The propositional formulas corresponding to $\mathcal{M}_{\rm f}$, $\mathcal{M}_{\rm c}$, $\mathcal{R}$, and $\mathcal{M}_{\rm fc}$, respectively, are:

$$\begin{array}{ll} M_{\rm f} \equiv f_1 \vee f_2 & R \quad \equiv (f_1 \Rightarrow c_1) \wedge (f_2 \Rightarrow c_2) \\ M_{\rm c} \equiv c_1 \Rightarrow c_2 & M_{\rm fc} \equiv M_{\rm f} \wedge M_{\rm c} \wedge R \end{array}$$

For which $\{f_2, c_2\}$ is an example of a model of the formula $M_{\rm fc}$, corresponding to the feature-component configuration $\langle\{f_2\}, \{c_2\}\rangle$.

6.1 On Implementing Propositional Feature-Component Models

When using models whose semantics is expressible in propositional logic, we have the advantage, over first-order logic for example, that the problems we are dealing with are typically decidable. We should note, however, that the complexity of many interesting problems remains a significant obstacle, e.g., consistency of a feature model is NP-complete [19].

We have implemented the ideas presented in this paper for the propositional logic case where the underlying data structure of the computations are binary decision diagrams (BDDs) [18]. The implementation is in an experimental stage and techniques that would make the approaches applicable in practice are under investigation. For the lack of space we cannot give the full account of the details but the reader is most welcome to contact the authors to obtain the source code. Here we briefly describe the main concepts used in the implementation.

Propositional induced feature models. Once the feature-component model is represented as a boolean expression, it is straight-forward to compute the induced feature model (see Def. 3) by applying the *existential quantification* [18, Sect 10.2.3]. Schematically, an existential quantifier from the formula $((\exists v)\phi)$ is eliminated by computing the formula $\phi[v \mapsto \mathit{true}] \vee \phi[v \mapsto \mathit{false}]$.

Propositional logic and strong conformity. Strong conforming configurations in the propositional case map directly to maximal models of a formula [14] by inverting the variables corresponding to components.

Realized-by expressions and their semantics. The realized-by expressions translate directly. For instance, realized-by$(\{\mathbf{f} \in \mathbb{F} \mid f_1 \in \mathbf{f}\}, \{\mathbf{c} \in \mathbb{C} \mid c_1 \in \mathbf{c}\})$ translates as realized-by(f_1, c_1), which in semantics $[\![\cdot]\!]_1$ is translated to $f_1 \Rightarrow c_1$.

6.2 Deriving Propositional Models

In practice, often, more complex constraints than propositional ones are needed. Here we wish to note that for certain types of constraints on attributes it is possible to generate an equivalent propositional constraint. We illustrate this idea on an example.

Example 8. Let $\text{size} : \mathcal{C} \rightarrow \mathbb{N}$ be a function assigning a size to each component. Let $l, h \in \mathbb{N}$ and the component model $\mathcal{M}_\text{c} \subseteq \mathcal{P}(\mathcal{C})$ be given in the form:

$$\mathbf{c} \in \mathcal{M}_\text{c} \Leftrightarrow \left(l \leq \sum_{c \in \mathbf{c}} \text{size}(c) \leq h\right)$$

Such a constraint can be translated into a propositional formula that disables the combinations of components whose total size is outside the given boundaries. Once this formula is computed, it can be "anded" to the formula obtained from the propositional feature-component model. Mathematically speaking, if a feature-component model is expressed by the formula M_fc and an additional constraint is expressed by the formula C, then we put $M'_\text{fc} \equiv M_\text{fc} \wedge C$ to impose the restriction C on M_fc.

7 Related Work

Kumbang [1] provides tool support for integrated feature and component modeling. The semantics of the modeling languages is defined in terms of the *weight constraint language*, which is supported by the `smodels` reasoning engine [20]. The reasoning is used during the configuration process, i.e., when certain selections are made, the engine is executed to infer the consequences of these selections. The reasoning of `smodels` is based on *stable models* semantics which guarantees that selections are not enforced without justification, which is similar to the component-minimality requirement in the semantics $[\![\cdot]\!]_3$ (see Sect. 5).

Thaker et al. [22] in the context of AHEAD Tool Suite [3] utilize a SAT solver to ensure that a feature model does not permit compositions that yield invalid Java programs (programs that do not type-check).

Van der Storm [23] maps features to components, which are organized in a dependency tree. Further, he maps this model to propositional logic semantics enabling reasoning on it.

Czarnecki and Pietroszek [9] investigated UML models with variability, so-called *model templates*, and utilized OCL to specify and the consistency requirements. These are utilized to check consistency of the models.

The works [22,23,9], described above, rely on propositional logic; Kumbang framework [1] enables more expressive constraints than propositional. We believe that all these works can be mapped to our formalism.

Automated reasoning was applied on feature models by a number of researchers. For example, Mannion applied Prolog to detect inconsistencies [16], Batory applied *logic truth maintenance systems* to facilitate the configuration process [2]. Benavides et al. investigated the use of constraint solving to automated analysis of feature models [5]. See [4] for a more complete overview.

McCarthy's *circumscription* [17] is a form of non-monotonic reasoning in principal similar to the semantics $[\![\cdot]\!]_3$ defined in Sect. 5. When using circumscription we are reasoning only with respect to minimal interpretation of a formula, similarly as in our approach we reason only about configurations that are strong conforming.

In Sect. 6.2 we have suggested how the techniques that reason on models expressible in propositional logic can be used to reason about other models that are not expressed as such. In a similar fashion, Eén and Sörensen preprocess so-called *pseudo-boolean constraints* inputted to a SAT solver [11].

8 Summary and Future Work

We have provided a unified mathematical approach to formalizing models for features and their implementations. We have shown how this formalization can be utilized to improve our understanding of the modeling primitives and provide automated feedback to the software engineer who uses such models.

We see the following challenges for future work.

Feedback to the user. If the induced feature model contains new derived dependencies not present in the original model, how should this "delta" be presented to the user? For this, we are investigating research by Czarnecki and Wąsowski [10] on turning boolean formulas to feature models.

Relating to languages used in practice. We have provided a formal foundations for expressing dependencies for integrated feature and component modeling. How can modeling languages used in practice be described using our formalism?

Other models. This work focuses on feature and component models. Could this approach be extended for an arbitrary number of models?

Implementation and Evaluation We will further improve our prototype implementation to evaluate the efficiency issues arising for larger numbers of features and to find out which of the semantics defined in Sect. 5 are useful in practice.

Acknowledgments We thank Ondřej Čepek for his valuable insight that improved the definition of strong conformity. This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303_1.

References

1. Timo Asikainen, Tomi Männistö, and Timo Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21, 2007.
2. Don Batory. Feature models, grammars, and propositional formulas. In H. Obbink and K. Pohl, editors, *Proceedings of the 9th International Software Product Line Conference (SPLC)*, 2005.
3. Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6), 2004.

4. David Benavides, Antonio Ruiz-Cortés, Pablo Trinidad, and Sergio Segura. A survey on the automated analyses of feture models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.
5. David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of 17th International Conference on Advanced Information Systems Engineering (CAiSE)*, 2005.
6. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison–Wesley Publishing Company, 2002.
7. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison Wesley, Reading, MA, USA, 2000.
8. Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In R. L. Nord, editor, *Proceedings of Third International Conference on Software Product Lines (SPLC)*, 2004.
9. Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 5th International Conference on Generative Programming And Component Engineering*, 2006.
10. Krzysztof Czarnecki and Andrzej Wąsowski. Feature diagrams and logics: There and back again. In Kellenberger [15].
11. Niklas Eén and Niklas Sörensen. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2006.
12. Mikoláš Janota and Joseph Kiniry. Reasoning about feature models in higher-order logic. In Kellenberger [15].
13. 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 CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, November 1990.
14. Dimitris J. Kavvadias, Martha Sideri, and Elias C. Stavropoulos. Generating all maximal models of a Boolean expression. *Information Processing Letters*, 74(3-4):157–162, 2000.
15. P. Kellenberger, editor. *Software Product Lines Conference*, 2007.
16. Mike Mannion. Using first-order logic for product line model validation. In G. J. Chastek, editor, *Proceedings of the Second International Conference on Software Product Lines (SPLC)*, 2002.
17. John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.
18. Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications*. Springer-Verlag, 1998.
19. 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)*, 2006.
20. Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2), 2002.
21. Software Engineering Institute. What is a software product line? http://www.sei.cmu.edu/productlines/.
22. Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE)*, 2007.
23. Tijs van der Storm. Generic feature-based software composition. In *Proceedings of 6th International Symposium on Software Composition*, 2007.