Modular Feature Models: Representation and Configuration

EBRAHIM BAGHERI, DRAGAN GASEVIC, FAEZEH ENSAN

Abstract. Within the realm of software product line engineering, feature modeling is one of the widely used techniques for modeling commonality as well as variability. Feature models incorporate the entire domain application configuration space, and are therefore developed collectively by teams of domain experts. In large scale industrial domains, feature models become too complex both in terms of maintenance and configuration. In order to make the maintenance and configuration of feature models feasible, we propose to modularize feature models based on the well established Distributed Description Logics formalism. Modular feature models provide for an enhanced collaborative/distributed feature model design, more efficient feature model evolution and better reusability of feature model structure. We also develop methods for the configuration and configuration verification of a modular feature model based on standard inference mechanisms. We describe and evaluate our proposed approach through a case study on an online electronic store application domain.

Keywords: Feature models, Variability Modeling, Software Product Lines, Modular Software Design

1. Introduction

A key aspect of software product line engineering is capturing the common characteristics of a set of software-intensive applications in a specific problem domain [18]. Product line engineering allows for the rapid development of variants of a domain specific application by using a common set of reusable assets often known as core assets. Such an approach supports the management of commonality as well as variability in the software development process [10, 11]. Feature modeling is an important conceptual tool that offers modeling support for software product lines. It provides for addressing commonality and variability both formally and graphically, describing interdependencies of the product family attributes (features) and expressing the permissible variants and configurations of the product family.

Feature modeling has received great attention both in industry and academia since its inception in the early 90’s by the Software Engineering Institute. Its concepts were developed within the Feature Oriented Domain Analysis (FODA) methodology [12]. In FODA, the distinguishing factors of the product family members are abstracted away until no difference between the members of the given problem domain exists. Other methods have been since developed on this basis such as Organization Domain Modeling (ODM) [14], FeatuRSEB [15], Feature-Oriented Reuse Method (FORM) [13] and cardinality-based feature modeling [10].

Different success stories of the use of product families and related techniques have been reported in the literature encompassing fields other than Computer Science such as the auto-manufacturing and cellular phone design industries. As an example, Clements and Northrop [1] reported that Nokia was able to increase its production capacity for new cellular phone models from 5-10 to around 30 per year. The main challenge that Nokia faced
was that as the market demand and customer taste changed rapidly, different products with varying specifications and user interfaces needed to be developed to keep up with the requests. The use of product family engineering made it possible to quickly and efficiently create the required software and hardware variations for the different products and use variability to customize the newly configured cellular phones. More recently, Microsoft has also become interested in software product families. It has incorporated the feature modeling process at the front end of its Software Factory life cycle to provide support for software product family engineering, which has been applied to its Health Level 7 (HL7) [2] software factory initiative. This initiative employs feature modeling to make the rapid design and development of HL7 conformant applications easier and more accessible to health informatics practitioners.

In complex and large domains, due to the complexity of the problems, feature models are often collaboratively developed by different teams. For instance, one of the scenarios for collaboration between the teams is that each team focuses on one set of conceptually-related features (e.g., one team focuses on developing all features related to the payment aspects of an online store, while another team focuses on developing the features related to shipping the sold items, etc.), then all the features developed by the different teams are merged together forming a comprehensive feature model. The other approach is that each team focuses on one specific set of goals of the product family and develops one feature model by identifying all of its required features regardless of their conceptual relevance (e.g., one team focuses on developing a feature model for easily deployable and cheap to install online stores, another team on a feature model for an online store that ensures highly secure and safe transactions, etc.). Once these feature models are developed, they can be put together to form one representative unique feature model. Therefore as it can be seen in these two scenarios, the required collaborative process for feature model development makes the maintenance of these models difficult to perform, which requires suitable representation mechanisms to store and integrate multiple feature models.

Furthermore, in light of the large-scale industrial applications of product families, the desire to create an overarching feature model for a product family entails the development of very large feature models that need to be customized before they can be used for a specific application in a given domain. In order to develop an instance of a product family from the relevant feature model, its most desirable features need to be selected from the feasible configuration space of the product family, whose entirety may not be of interest for a specific application at hand. The selection of the best set of features for a product would be based on the strategic goals, requirements and limitations of the stakeholders, as well as the integrity constraints of the feature model. Once the desired features are specified, the feature model can be customized such that it includes the wanted features and excludes the non-relevant ones. A final fully-specific feature model with no points for further customization is called a configuration of the feature model based on the selected features. In many cases, a configuration is gradually developed in several stages.

In each stage, a subset of the preferred features are selected and finalized and the unnecessary features are discarded yielding a new feature model whose set of possible configurations are a subset of the original feature model. This feature model is referred to as a specialization of the former feature model and the staged refinement process constitutes staged configuration [3].
The adopted representation mechanism that enhances feature model maintenance should also provide means for efficient specialization, configuration and validation of feature models.

1.1. Problem Description

As mentioned earlier, there are several challenges towards the maintenance and specialization of large-scale feature models. In this paper, we are specifically looking at addressing the following problem statements:

1. Given a complex target domain with a large feature space that demands the involvement of multiple experts, what would be the most appropriate feature model representation mechanism such that expert collaboration, and feature model maintenance is facilitated. For example, in the context of the two mentioned feature model development collaboration scenarios, what would be an acceptable approach for the representation of feature models in such a way that experts can efficiently perform their tasks individually and also share feature model information collectively;

2. The specialization of feature models is performed based on the requirements of the stakeholders and the target application. In many cases, the stakeholders only specify a limited number of required features and expect a complete configuration of the feature model accordingly. Since the configuration needs to be developed based on feature interactions, feature model integrity constraints and the requested stakeholder requirements, what would be an appropriate method in the context of the developed representation discussed above for finding a suitable feature model configuration that satisfies the stakeholders.

The first issue is related to the maintenance of large-scale feature models, while the latter is concerned with their manipulation and specialization.

1.2. Approach Overview

In order to tackle to the two mentioned challenges, we take a holistic approach where both problems are viewed in a unified framework. Fundamentally, we adopt Description Logics [4] as the underlying formalism for representing feature models, and strategize in this context such that appropriate techniques are developed for our purpose. Due to their advantages discussed below, modularity and concept covering lay at the heart of our framework:

1. **Modularity** is an engineering technique that builds larger systems by combining smaller components. A modular design allows for the rise of important design attributes such as the increase in extendability, reusability, parallel and collaborative design progression and others. Such an approach in feature model development extends our ability for decomposing the problem domain into smaller subdomains that can be analyzed and modeled simultaneously, while also providing for the unification of these subdomains later. To support for collaborative feature model design through modularity, we
EBRAHIM BAGHERI, ET AL

customize the Distributed Description Logic framework to represent modular feature models;

2. Concept covering is a computational minimization process that evaluates whether a certain conceptual structure wraps around another, or how extended the structure should be to be able to do that. We provide a correspondence between the concept covering problem in Description Logics [5] and feature model configuration and show how an acceptable configuration can be developed based on the feature model integrity constraints and the stakeholder requirements. We further demonstrate how concept covering can be beneficial in pinpointing to the stakeholder requests that are hampering the development of a suitable feature model configuration, when one cannot be developed.

In summary, we are interested in looking at how large and complex feature models can be maintained through an appropriate representation mechanism, and furthermore how these large feature models can be configured. For the first issue, we explore the possibility of feature model modularization in Description Logics, in the context of which we will benefit from existing concept covering methods to create appropriate configurations for large-scale feature models.

The remainder of this paper is organized as follows: Section 2 provides an overview of the main concepts of feature modeling along with an introduction to the sample online electronic store feature model. The representation of feature modeling notations in Description Logics is discussed in Section 3. The modular representation of feature models in Distributed Description Logics is shown in Section 4, followed by the formalization of the feature model configuration process through concept covering. In Section 6, two case studies are presented. Section 7 provides some discussions. In Section 8 elaboration on the related work is provided. Finally, the paper is concluded in Section 9.

2. Feature Modeling

Features are important distinguishing aspects, qualities, or characteristics of a family of systems [6]. They are widely used for depicting the shared structure and behavior of a set of similar systems. To form a product family, all the various features of a set of similar/related systems are composed into a feature model. A feature model is a means for representing the possible configuration space of all the products of a system product family in terms of its features. Feature models can be represented both formally and graphically; however, the graphical notation depicted through a tree structure is more favored due to its visual appeal and easier understanding. More specifically, graphical feature models are in the form of a tree-like structure whose root node represents a domain concept, e.g., a domain application, and the other nodes and leaves illustrate the features. In this context, a feature is a concept property related to a user-visible functional or non-functional requirement, e.g., domain application task, modeled in a way to capture commonalities or possibly differentiate among product family variants.

2.1. Structure of Feature Models

In a feature model, features are hierarchically organized and can be typically classified as:
MODULAR FEATURE MODELS: REPRESENTATION AND CONFIGURATION

Figure 1. The main graphical notations employed in feature modeling.

- **Mandatory**, the feature must be included in the description of its parent feature;
- **Optional**, the feature may or may not be included in its parent description given the situation;
- **Alternative feature group**, one and only one of the features from the feature group can be included in the parent description;
- **Or feature group**, one or more features from the feature group can be included in the description of the parent feature.

Figure 1 depicts the graphical notation of the feature relationships. The tree structure of feature models falls short at fully representing the complete set of mutual interdependencies of features; therefore, additional constraints are often added to feature models and are referred to as **Integrity Constraints (IC)**. The two most widely used integrity constraints are:

- **Includes**, the presence of a given feature (set of features) requires the **existence** of another feature (set of features);
- **Excludes**, the presence of a given feature (set of features) requires the **elimination** of another feature (set of features).

In addition to this information, several feature modeling notations such as the cardinality-based feature models [7] provide means for expressing the plausible cardinality of the features.

2.2. The Online Electronic Store Feature Model

The online store is a commonly used example in the feature modeling literature, which is employed to represent the behavior of feature modeling frameworks [8]. Figure 2 depicts a small feature model designed to depict some of the aspects of such an online store. The sample electronic store consists of three mandatory features, namely: Payment, Tax, and Password, and one optional feature: Shipping. Furthermore, the Shipping feature has been represented by an or-group of sub-features, while an alternative-group of sub-features constitutes the Tax feature. It is easy to see that with this design, various configurations for an electronic store can be developed. For instance, one store may decide to use airmail to ship its goods, while another can only provide pickup service to its customers.
Clearly, not all possible configurations of the features of e-store produce valid electronic store programs. For instance, an automatic fraud detection feature would not be so helpful while the payments are made in the form of cash. Therefore, not all possible configurations of the feature model are valid and some restrictions on the acceptable feature configurations exists. Such restrictions are expressed in the form of feature model integrity constraints. Several examples of such integrity constraints for the online electronic store feature model are: 1) cash payment includes pickup shipping; 2) credit card payment includes automatic fraud detection; 3) never changing the password includes the inclusion of special characters in the password; and 4) manual fraud detection excludes credit card payment; 5) e-delivery excludes credit card payment; 6) credit card payment includes special character passwords; and 7) pickup excludes universal tax rate.

Now based on the structure of the feature model and its integrity constraints, valid feature model configurations can be developed and validated.

3. Feature Model Representation in Description Logics

Our proposed formal framework borrows from Semantic Web languages and particularly Description Logics [4]. Description Logics (DLs) are a family of logic formalisms for knowledge representation, whose basic syntax elements are concept names, role names, and individuals. Intuitively, concepts stand for sets of objects in the domain, and roles link
objects in different concepts. Individuals are used for special named elements belonging
to concepts. Formally, concepts are interpreted as subsets of a domain of interpretation $\Delta$, and roles as binary relations, which are subsets of $\Delta \times \Delta$.

DL formulas (concepts) are given semantics by defining the interpretation function over
each construct. For example, given two generic concepts such as $C$ and $D$, their conjunc-
tion $C \cap D$ is interpreted as set intersection: $(C \cap D)^I = C^I \cap D^I$, and also the other boolean connectives are given the usual set-theoretic interpretation, whenever present. The interpretation of constructs involving quantification on roles needs to make domain el-
ments explicit. Concepts can be used in inclusion assertions $C \subseteq D$, and definitions
$C \equiv D$, which pose restrictions on possible interpretations according to the knowledge elicited for a given domain. The semantics of inclusions and definitions is based on set contain-
ment: an interpretation $I$ satisfies an inclusion $C \subseteq D$ if $C^I \subseteq D^I$, and satisfies a
definition $C \equiv D$ when $C^I = D^I$. A DL theory, also known as TBox or ontology, is a set
of inclusion assertions and definitions. A model of a TBox $T$ is an interpretation satisfying all inclusions and definitions of $T$. It is possible to introduce a variety of constructors in
DL languages, making them more expressive. However, it is well known that this may lead
to an explosion in computational complexity of inference mechanisms. Hence, a trade-off
is necessary. In this paper, we refer to $\mathcal{ALC}$ (the logic $\mathcal{AL}$ plus complex concept negation).

Given the syntax and semantics of $\mathcal{ALC}$, we are able to model the structure and integrity
constraints of feature models in the $\mathcal{ALC}$ Description Logics, hence it is expressive-enough
for our purpose. In the following similar to [8], we show how the structural knowledge,
the integrity constraints and the stakeholder requests can be represented in the selected DL
language.

3.1. Structural Knowledge

The structural knowledge base, denoted $\mathcal{SKB}$, consists of the description of the hierarchi-
cal feature relationships of a feature model. The fundamental feature hierarchy introduced
in Section 2 is expressed in this knowledge base. As base constructs, we define $\text{Feature}$
as the parent of all available features, and also let $\text{hasFeature}$ be the core role that relates
parent features with their child features.

$$\begin{align*}
\text{Feature} & \subseteq \top \\
\text{hasFeature} & \subseteq \text{ObjectProperty}
\end{align*}$$

**Mandatory** a mandatory feature can be viewed as a prerequisite for the existence of its
parent feature;

$$\begin{align*}
\text{Child} & \subseteq \text{Feature} \\
\text{Parent} & \subseteq \text{Feature} \\
\text{Parent} & \subseteq \exists \text{hasFeature}.\text{Child}
\end{align*}$$

In case multiple features are mandatory, they can be formulated by

$$\begin{align*}
\text{Child}_i & \subseteq \text{Feature}, \quad \text{for } 1 \leq i \leq n \\
\text{Parent} & \subseteq \text{Feature} \\
\text{Parent} & \subseteq (\exists \text{hasFeature}.\text{Child}_1) \cap \ldots \cap (\exists \text{hasFeature}.\text{Child}_n)
\end{align*}$$
which means that all of the child features should be present in the configuration of the parent feature.

**Optional** optional features are free to appear or not in the formation of the parent feature; hence, both states of being false or true are acceptable and therefore, does not require their direct connection with their parent features;

\[ \text{Child} \sqsubseteq \text{Feature} \]

however, it should be noted that due to the nature of optional features and the fact that they may not be present in the configuration of their parent feature, they often appear in conjunction with other sibling features.

**Or feature group** an or feature group allows the feature parent to become configured with the appearance of at least one of its child features:

\[
\begin{align*}
\text{Child}_i & \sqsubseteq \text{Feature}, & & \text{for } 1 \leq i \leq n \\
\text{Parent} & \sqsubseteq \text{Feature} \\
\text{Parent} & \sqsubseteq (\exists \text{hasFeature.Child}_1) \sqcup \ldots \sqcup (\exists \text{hasFeature.Child}_n)
\end{align*}
\]

**Alternative feature group** in an alternative feature group, features are represented in a mutually exclusive form, where the appearance of one feature eliminates the possibility of the other. Essentially, the logical relationship between the features is similar to the one we used for or feature groups with the addition of mutually disjunction axioms between siblings in the tree

\[
\begin{align*}
\text{Child}_i & \sqsubseteq \text{Feature}, & & \text{for } 1 \leq i \leq n \\
\text{Child}_i & \sqsubseteq \neg \text{Child}_j, & & \text{for } 1 \leq i, j \leq n \land i \neq j \\
\text{Parent} & \sqsubseteq \text{Feature} \\
\text{Parent} & \sqsubseteq (\exists \text{hasFeature.Child}_1) \sqcup \ldots \sqcup (\exists \text{hasFeature.Child}_n)
\end{align*}
\]

This formulation shows that the elements of an alternative feature group cannot co-exist in the configuration of their parent feature.

More complex clauses can be formulated based on these primitive relationships. As an example, the shipping feature shown in Figure 2 can be represented using an or feature group:

\[
\begin{align*}
\text{Shopping} & \sqsubseteq \text{Feature}, \text{Pickup} \sqsubseteq \text{Feature}, \text{E – delivery} \sqsubseteq \text{Feature} \\
\text{Airmail} & \sqsubseteq \text{Feature}, \text{RegularMail} \sqsubseteq \text{Feature} \\
\text{Shopping} & \sqsubseteq (\exists \text{hasFeature.Pickup}) \sqcup \ldots \sqcup (\exists \text{hasFeature.RegularMail})
\end{align*}
\]

As another example, the fraud detection feature can be modeled as:

\[
\begin{align*}
\text{FraudDetection} & \sqsubseteq \text{Feature}, \text{Manual} \sqsubseteq \text{Feature}, \text{Automatic} \sqsubseteq \text{Feature}, \\
\text{Manual} & \sqsubseteq \neg \text{Automatic} \\
\text{FraudDetection} & \sqsubseteq (\exists \text{hasFeature.Manual}) \sqcup (\exists \text{hasFeature.Automatic})
\end{align*}
\]
Finally, for the feature model root in addition to its feature relationships, we define the root as:

\[
\text{root} \sqsubseteq \exists \text{hasFeature.Feature}
\]

For instance, the E-Store feature model root is represented as:

\[
E \text{- Store} \sqsubseteq \exists \text{hasFeature.Tax} \sqcap \exists \text{hasFeature.Password} \sqcap \exists \text{hasFeature.Payment}
\]

The information represented in this form are referred to as structural knowledge of the feature model and will be stored and referred to as SKB.

### 3.2. Integrity Constraints

Feature interdependencies that cannot be captured in structured hierarchical format are represented within the Integrity Constraints base (IC).

**Includes**  the presence of a given feature could entail the automatic inclusion (the need for presence) of one or more other features, which is represented as:

\[
\exists \text{hasFeature.BaseFeature} \sqsubseteq \exists \text{hasFeature.IncludedFeature}
\]

For instance, cash payment includes pickup shipping can be represented as

\[
\exists \text{hasFeature.Cash} \sqsubseteq \exists \text{hasFeature.Pickup}
\]

**Excludes**  the inclusion of one feature may result in the exclusion of one or more features, which is shown by:

\[
\exists \text{hasFeature.BaseFeature} \sqsubseteq \neg(\exists \text{hasFeature.IncludedFeature})
\]

For instance, manual fraud detection excludes credit card payment would be depicted as:

\[
\exists \text{hasFeature.Manual} \sqsubseteq \neg(\exists \text{hasFeature.CreditCard})
\]

All integrity constraints will be stored and represented through the integrity constraint base (IC).

### 3.3. Stakeholder Requests

Stakeholders and product developers can specify a set of basic features that they would like to see in the final product. For instance, in the e-store feature model configuration process, they may require the inclusion or exclusion of certain features. Such requirements and requests are referred to as the stakeholders’ hard constraints. The satisfaction of hard constraints is either feasible or not, which makes the configuration process based
on hard constraints a crisp one. Hard constraint can be represented by complex concepts in Description Logic format. For instance, a request by the stakeholders to have both air-mail delivery and credit card payment features available can be represented by a concept description defined as: $\text{Airmail} \cap \text{CreditCard}$. We represent the stakeholders’ hard constraints through $SR_h$.

Summing up with regards to the knowledge domain modeling we have the following formalization:

$SKB$: The feature model structural knowledge is represented using DL concepts and role relations modeled as described in Section 3.1.

$IC$: Integrity constraints are subsumption relations as described in Section 3.2.

$SR_h$: Stakeholder’s hard constraints are complex concept descriptions explained above.

4. Modularity for Feature Models

In this section, we first show how the collaboration scenarios for developing feature models can produce multiple feature models and hence require a modular design. Based on this, the DDL framework is customized to represent modular feature models.
4.1. **Collaborative Feature Model Design**

As was discussed earlier, large domains require the collaboration of several teams of modeling and domain experts to fully cover the wide range of available features. The collaborative process structure can be defined based on how the feature model is to be developed. In the following we introduce two sample collaboration scenarios:

1. It is possible that in a complex domain, some of the required applications have already been analyzed and legacy software applications exist. For this reason, feature models can be more easily developed for the segments of the target domain which already possess a running application or a designed conceptual model. In case several applications of that product family exist, teams of experts can be assembled to focus on each one of these existing applications in order to develop the corresponding feature models for these applications. Later, it is possible to integrate each of the developed individual feature models by unifying their common features and incorporating their differences into one representative feature model. Figure 3 shows this case where multiple existing fragmented feature models for the domain of e-store have been developed by different teams of experts. As it can be seen in this figure, each individual feature model can
encompass all of the aspects of the domain; however, it may not be complete in all of the aspects.

2. The second possible form of collaboration is shown in Figure 4. In this approach, the target domain is divided into non-overlapping subdomains, based on which different teams of experts are assembled such that each of the teams focuses on one specific subdomain of interest. The outcome of the work of each team would be a complete subdomain analysis and its corresponding feature model design. As shown in Figure 4, the individual feature models do not have any overlap with each other. The integration of these feature models will entail a complete feature representation for the domain.

In both collaboration scenarios, multiple individual feature models are developed by the individual teams. So the main challenge is to interrelate the individual feature models such that they form a meaningful and comprehensive feature model. In other words, each of the individual feature models can be viewed as an independent module of a larger feature model; therefore, the large feature model constitutes several individual feature models and their interrelationships. This approach allows for a modular approach towards feature model design.

4.2. Modular Feature Models

Given the set of individual feature models developed individually by different teams, it is important to investigate how these feature models, referred to as feature model modules, are interrelated and integrated in terms of their integrity constraints and feature relationships such that they form one representative whole. We base our formalization on the notions in DDL. In the following, we would like to show how individual modules can integrated such that they form a large modular feature model.

**Definition 1** Given a set \( F = \{F_i\}_{i \in I} \) of feature model modules represented in ALC Description Logic, we denote the TBox of \( F_i \) as \( T_i \). Each TBox contains all the required structural knowledge as well as the integrity constraints of \( F_i \), i.e., \( T_i = \text{SKB}_i \sqcup \text{IC}_i \). In this context, the set of all feature model modules can be represented as a corresponding family of self-consistent TBoxes \( T = \{T_i\}_{i \in I} \).

In order to unambiguously refer to the features in different feature modules, they are prefixed with the index of the feature module, e.g., \( i : \text{Pickup} \sqsubseteq \text{Feature} \) shows that the pickup feature has been defined in feature module \( i \) (i.e., \( i \) denotes the index of the feature module from which the feature, Pickup, has been derived).

Furthermore, to interrelate the individual feature modules, bridge rules in DDL are employed, and are referred to as module bridges.

**Definition 2** Given a family of feature model modules and their corresponding TBoxes \( T = \{T_i\}_{i \in I} \), a module bridge from module \( i \) to \( j \) is a feature model integrity constraint of one of the following forms:

\[
\begin{align*}
&i : X \xrightarrow{\Delta} j : Y, \text{ an into-module bridge;} \\
&i : X \xrightarrow{\nabla} j : Y, \text{ an onto-module bridge.}
\end{align*}
\]
where $X$ and $Y$ are feature model structural knowledge from $T_i$ and $T_j$, respectively. In case, one specific feature is represented differently in two feature modules, an equivalence-module bridge can be derived as a pair of into- and onto-module bridges, denoted $i : X \equiv j : Y$ to show that the features refer to the same concept.

**Example 1** Let us consider the e-store feature model represented through four feature model modules in Figure 4. Now, recalling that features in different modules can be interrelated through integrity constraints, their relationships can be represented as module bridges. The following are the possible examples of module bridges:

1 : $\exists \text{hasFeature.Cash} \rightarrow 3 : \exists \text{hasFeature.Pickup}$
3 : $\exists \text{hasFeature.E - delivery} \rightarrow 1 : \exists \text{hasFeature.CreditCard}$
1 : $\exists \text{hasFeature.CreditCard} \rightarrow 4 : \exists \text{hasFeature.SpecialCharPassword}$
2 : $\neg(\exists \text{hasFeature.UniversalTaxRate}) \rightarrow 3 : \exists \text{hasFeature.Pickup}$

Module bridges allow for the integration of different feature modules through the application of integrity constraints as onto- or into-module bridges. Simply stated, module bridges can be viewed as integrity constraints between the features of two different feature modules. It is also possible to relate duplicate features that are present in multiple feature modules through equivalence-module bridges.

Given the notion of module bridges, it is possible to define modular feature models as a collection of feature modules and their interrelating module bridges. To this end, we define a feature model sentinel as a feature module that constitutes the core feature modeling concepts and feature model configurations of a modular feature model and unifies all of the individual feature modules into a whole.

**Definition 3** A feature model sentinel is a feature module $S$, which is connected pair-wise to all feature modules, and consists at least of the following feature description concepts and roles:

$\text{GenericFeature} \sqsubseteq \top$

$\text{hasGenericFeature} \sqsubseteq \text{ObjectProperty}$

$\text{GenericProductLine} \sqsubseteq \exists \text{hasGenericFeature.GenericFeature}$

$\text{OurProductLine} \sqsubseteq \text{GenericProductLine}$

A feature model sentinel serves as the backbone of a modular feature model by containing the basic required axioms for defining a feature model. Feature definitions in the individual feature modules are extensions of these base concepts included in the sentinel. Later, we will show that stakeholder requests, $SR_h$, and any developed feature model configuration will be stored in the feature model sentinel. Interestingly, the implication of a sentinel is that it makes the individual feature modules independent of the final modular feature model that they are going to appear in. For instance, given a set of five feature modules, one can create a sentinel to form a modular feature model from three of them.
Figure 5. The topology of the modular feature model of Figure 4: a) the module bridge graph b) the adjacency matrix for the weighted module bridge graph.

to address some problem domain of interest, while another person can create a different sentinel that brings all five of these modules into play to create a completely different modular feature model and addresses a different problem. In essence, sentinel is for modular feature models what glue code is for component-based software development.

Generally speaking, a feature model sentinel is the connecting module that shapes a modular feature model defined as follows:

Definition 4 A modular feature model is a triple \((F, B, S)\) where \(F = \{F_i\}_{i \in I}\) represents the individual feature modules, \(B = \{B_{ij}\}_{i \neq j \in I}\) denotes the set of all available module bridges, and \(S\) is the feature model sentinel.

In order to characterize the topology of a modular feature model, a corresponding module bridge graph can be developed.

Definition 5 Given a modular feature model \((F, B, S)\), a module bridge graph is a directed graph with a set of nodes corresponding to the collection of \(F = \{F_i\}_{i \in I}\) and an arc from node \(i\) to node \(j\) when the set of module bridges \(B_{ij} \in B\).

Although \(B_{ii}\) is not permitted in a modular feature model, we allow it to be present in the module bridge graph. The module bridge graph can be extended such that its arcs represent the number of module bridges between the connected nodes yielding a weighted module bridge graph. Module bridge graphs are appropriate underlying data structures for defining and analyzing structural quality metrics for feature models, such as cohesion, coupling, reusability, and others. Further analysis of module bridge graphs are outside the scope of this paper and will be discussed in separate work. Figure 5 shows the module bridge graph and its corresponding adjacency matrix for the modular feature model depicted in Figure 4.

Example 2 Let us consider the feature modules depicted in Figure 4. A modular feature model developed from these four individual feature modules consists of a feature model
Figure 6. A modular feature model with four feature modules and a sentinel.

As it can be seen in Figure 6, the sentinel sits at the center of the modular feature model and provides the base required constructs. The sentinel is connected to all feature modules through several module bridges, denoted as the set $B_S$. Each of the modules between the sentinel and the other feature modules ($B_S$) consists of the following axioms (not shown in the figure, in other words the arrow labeled $B_S$ will be carrying the following axioms):
Also the sentinel is given access to the top feature(s) of each module through a module bridge \((B_S)\) in order for us to be able to build the overall structure of the modular feature model in the sentinel. For instance, \(B_S)\) consists of \((S : Payment \rightarrow 1 : Payment)\), which shows that the payment feature being referred to in the sentinel is equivalent to that defined in feature module 1. With the presence of these module bridges, a complete feature model can be put together in the sentinel for e-store as follows:

\[
E = \text{store} \sqsubseteq \exists\text{hasGenericFeature.Payment} \sqcap \exists\text{hasGenericFeature.Tax} \sqcap \exists\text{hasGenericFeature.Password}
\]

\(\text{Shipping} \sqsubseteq \text{GenericFeature}\)

Note that the features such as Payment, Tax, Password, and Shipping have already been defined in their respective feature modules and have been imported to the sentinel through the module bridges. So, the sentinel and the module bridges form a complete integration of individual feature modules and create a structured modular feature model for the e-store domain.

It is important to mention that the design teams responsible for each of the individual feature modules need to come together to designate the required module bridges in order to have a nice integration of the feature modules. The development and representation of feature models in a modular manner provides many benefits and advantages, such as:

- **Distributed/collaborative development:** As mentioned earlier, modularity will enhance the possibility of collaboration between several design and development teams. Individual feature modules can be created by different teams and easily integrated as a constituting element into the comprehensive feature model;

- **Feature model evolution:** While the domain dynamics gradually change throughout time, the structure and available features of the feature model need to be updated and changed accordingly. A modular feature model allows for a smoother change process where feature modules can be updated or even replaced without a lot of side-effects;

- **Module reusability:** The development of structured segments for a feature model in terms of modules allows domain analysts and feature model designers to reuse the developed modules in other appropriate feature modeling processes. This increases productivity, promotes knowledge transfer between projects, and reduces the likelihood of repeating design mistakes;

In the following, we investigate how a given configuration of a modular feature model can be validated to see whether they conform to its structural knowledge and integrity constraints.
4.3. Configuration Verification

A configuration of a feature model is the result of the selection of a certain number of feature model elements and the removal of the others. A feature model configuration corresponds with the requirements of a specific domain application. Designers often configure a feature model such that the maximum number of stakeholder requests and requirements for the given application are satisfied; however, in large feature models, due to the structural and integrity constraints imposed on the available features, not all configurations are valid. With the increase in the number of selected features, keeping track of the valid configurations becomes harder to be performed manually. In this paper, since the representation of modular feature models is performed in Description Logics, it is possible to use standard Description Logic inference mechanisms to perform the configuration validation process. Any Description Logic-based system provides at least two basic standard reasoning mechanisms:

Concept satisfiability  \( T \models D \not\subseteq \bot \). Given a DL TBox \( T \) and a concept \( D \), does there exist at least one model of \( T \) assigning non-empty extension to \( D \)?

Subsumption  \( T \models D \subseteq C \). Given a DL TBox \( T \) and two concepts \( C \) and \( D \), is \( D \) more general than \( C \) in any model of \( T \)?

So, given a feature model \( F \) expressed in terms of Description Logic TBox \( T \), which conveys a software product line description, such as that of the electronic online store, \( PL = SKB \sqcup IC \), and a feature model configuration \( C \), using the two standard reasoning mechanisms it is possible to see whether 1) the feature model configuration fully conforms with the feature model description \( T \models C \subseteq PL \), which is referred to as the entailment of concept subsumption; 2) they are at least compatible with each other but the configuration may contain features that are not included in the feature model \( T \models PL \cap C \not\subseteq \bot \), which is called the satisfiability of concept conjunction; 3) the feature model configuration is not at all compatible with the feature model \( T \models PL \cap C \subseteq \bot \), referred to as the entailment of concept disjointness. Hence, it is quite straightforward to verify whether a feature model configuration developed manually is a valid configuration or not. However, since concept satisfiability and subsumption perform on a monolithic TBox, we need to temporarily convert our modular feature model representations into a monolithic representation. Distributed Description Logics provides the means to do so through the bridge operator.

Definition 6  Given a set of module bridges \( B_{ij} \) from feature module \( F_i \) to \( F_j \), an operator \( B_{ij}(\cdot) \), taking as input features in \( F_i \) and producing a set of corresponding features for them in \( F_j \) is called the bridge operator.

Details of the bridge operator can be found in [9]. For our purpose, we need to perform the bridge operator on and from all available feature modules onto the feature model sentinel. In this way, the sentinel will temporarily contain the representative monolithic form of the modular feature model, over which the feature model configuration verification process can be easily performed. Formally stated, feature model configuration verification can
be performed as:

$$\bigcup_{i} B_{i} S(\cdot) \cup S \models C \subseteq \text{PL}$$

where $B_{i} S(\cdot)$ denotes the application of the bridge operator from all the feature modules onto the sentinel, and $i$ iterates over the available feature modules. With this standard DL reasoning mechanism, designers can easily validate their developed feature model configurations.

It is important to mention that from a high level designer perspective the verification process is performed on the modular form of the feature models, and the designers are oblivious to the temporary transformation of feature modules into a temporary monolithic sentinel. This is also true for the algorithms proposed in the following sections, where the designers interacting with the feature model will only see a modular feature model representation, which preserves all of its discussed benefits.

In the following section, we will benefit from and customize a method originally proposed by Ragone et al. for distributed Web service composition [10]. In their method, concept abduction is used to address the concept covering problem in Description Logic, which is employed for creating an automated Web service composition process. As we will see, their approach provides inexact composition solutions when an exact solution cannot be found. We benefit from the similarity of the Web service composition problem and modular feature model configuration, in which both problems are trying to create a customization for a set of distributed lowly coupled entities based on some given criteria, to create a configuration process for modular feature models.

5. Modular Feature Model Configuration

It is common practice that feature model configurations are derived based on stakeholders’ requests, i.e., the stakeholders specify the set of important features that are required to be present in the final product. Based on these stakeholder requests, the designers would then try to customize the feature model such that the maximum number of these requests are satisfied. However due to feature interdependencies, it is possible that not all of the requests are satisfiable. Furthermore, the complexity of large industrial-scale feature models makes it hard for the designers to manually find an acceptable configuration for the feature model based on the requests; therefore, the development of automated feature model configuration techniques that would facilitate this process is desirable. For instance, it would be beneficial to have an automated configuration tool that would take as input the stakeholder requests, the structural knowledge and the integrity constraints of the feature model and provide as output a suggested configuration. Given the suggested configuration, the designers and the stakeholders can evaluate the desirability of the configuration, and interactively change their requirements until an appropriate configuration is developed.

In order to develop such an automatic modular feature model configuration technique, we base our work on the notion of concept covering as presented in [10]. The following introduces how concept covering in Description Logics can be employed to perform feature model configurations in modular feature models.
5.1. The Concept Covering Problem

In the context of the modular feature model configuration verification process, it was shown that concept subsumption can be used to evaluate whether a given feature configuration, denoted \( C \), is consistent with a feature model description, denoted \( PL \). In fact if the relation \( T \models \{ C \subseteq PL \} \) holds, the configuration is consistent within the framework of the feature model; however, if \( T \models \{ PL \cap C \subseteq \bot \} \) holds, since satisfiability and subsumption only return a boolean truthfulness value, it is not possible to find an explanation for why the entailment of concept disjointness holds. The Concept Abduction Problem (CAP) was introduced and defined in [5], to provide an explanation hypothesis when subsumption does not hold.

**Definition 7** Let \( S, D \) be two concepts and \( T \) a set of axioms in a Description Logic \( L \), where both \( S \) and \( D \) are satisfiable in \( T \). A CAP, denoted as \( \langle L, S, D, T \rangle \), is finding a concept \( H \) such that \( T \not\models \{ S \cap H \subseteq \bot \} \), and \( T \models \{ S \cap H \subseteq D \} \).

Given a CAP, if \( H \) is a conjunction of concepts and no sub-conjunction of concepts in \( H \) is a solution to the CAP, then \( H \) is an irreducible solution. In [5], the minimality criteria for \( H \) and a polynomial algorithm, to find solutions which are irreducible, for \( ALC \) DL, have been proposed. More recent work has extended support for CAP in \( SH \) DL [11], which is an extension of \( ALC \) DL that we use in this paper (\( ALC \subseteq SH \)) and can hence be applied to our specific problem.

As described in [12], the solution to a CAP can be read as *why does not \( S \subseteq D \) hold?*, what is a possible explanation with respect to \( T \)? In other words \( H \) represents what is expressed, explicitly or implicitly (that is, entailed because of the TBox), in \( D \) and is not present in \( S \), or also *which part of \( D \) is not covered by \( S \)*?

So, how can a solution to CAP be useful for modular feature model configuration? To see the relationship, let us suppose that stakeholder requests are represented as \( SR_h \), and some feature model configuration \( C \) has been provided by the designers to satisfy the stakeholders’ requests. It is intuitive that \( H \), in a CAP solution, is an explanation hypothesis for the missing part in the available feature model configuration \( C \) needed to completely satisfy a stakeholders’ requests in \( SR_h \). A not-full match with the stakeholders’ request is then due to \( H \). This is an observation that can also be expressed by defining \( H \) as: what is not covered by \( C \) with respect to \( SR_h \).

Based on this last remark, we use solutions to sets of CAP to create a correspondence between modular feature model configuration and concept covering and solve concept covering problems as it is shown in the following. In particular, we show that we are able, through concept abduction, to formulate the definition of concept covering, and we introduce a greedy algorithm\(^2\) to compute concept covering, or in other words feature model configurations based on stakeholders’ requests, in terms of concept abduction [13, 14]. Such an algorithm provides as output the obtained feature model configuration and, when such a complete configuration does not exist, a logical explanation on what remains missing in the provided feature model configuration with regards to the stakeholders’ requests.

Here, we use the following definition for the concept covering problem:
Definition 8 ([10]) Let $D$ be a concept, $R = \{S_1, S_2, \ldots, S_k\}$ be a set of concepts, and $T$ be a set of axioms, all in some DL $\mathcal{L}$, e.g., $\mathcal{L}=\text{ALC}$, where $D$ and $S_1, \ldots, S_k$ are satisfiable in $T$. Let also $\prec_T$ be an order relation over $\mathcal{L}$ taking into account the TBox $T$. The Concept Covering Problem (CCoP) denoted as $\forall=(\mathcal{L}, R, D, T)$ is finding a pair $(R_c, H)$ such that 
1. $R_c \subseteq R$, and $T \not\models \cap_{S \in R_c} S \equiv \bot$; 
2. $H$ is a solution to $(\mathcal{L}, S, D, T)$, and $H \prec_T D$.

We call $(R_c, H)$ a solution for $\forall$, and say that $R_c$ covers $D$. Finally, we denote $\text{SOLCCoP}(\forall)$ the set of all solutions to a CCoP $\forall$.

In the above definition, the elements for the solution $(R_c, H)$ of a CCoP represent respectively:

- $R_c$: Which concepts in $R$ represent the cover for $D$ w.r.t. $T$.
- $H$: What is still in $D$ and is not covered by concepts in $R_c$.

Intuitively, $R_c$ is a set of concepts that completely or partially cover $D$ w.r.t. $T$, while the abduced concept $H$ represents what is still in $D$ and is not covered by $R_c$.

Di Cugno et al. [15] argue that concept covering problem is similar, but has remarkable differences when compared to classical set covering. CCoP is not trivially a different formulation of a classical minimal set covering ($\text{SC}$) problem, as an exact solution to a CCoP may not exist. Furthermore in SC elements are not related with each other, while in CCoP elements are related with each other through axioms within the TBox, and while the aim of an SC is to minimize the cardinality of $R_c$, the aim of a CCoP is to maximize the covering, hence minimizing $H$.

Several solutions can exist for a single CCoP, depending on the strategy that is adopted for choosing concepts present in $R_c$. This fact gives way to the definition of best cover and exact cover.

Definition 9 ([16]) A best cover for a CCoP $\forall$, w.r.t. an order $\prec_T$ on the adopted DL, is a solution $(R_c, H_b)$ for $\forall$ such that there is no other solution $(R'_c, H'_b)$ for $\forall$ with $H'_b \prec_T H_b$.

There is no solution $(R'_c, H'_b)$ for $\forall$ such that $H'_b$, the remaining part of $D$ still to be covered, is ‘smaller’ than $H_b$ with respect to an order $\prec_T$. Observe that, since the solution for a concept abduction is not unique in general, there could be two solutions $(R_c, H_1), (R_c, H_2)$ such that the first is a best cover, while the second one is not. However, when a full cover exists, it is independent of any order $\prec_T$ on the adopted DL. However, for our purpose the SIM-DL algorithm presented in [17] can be used to compute an order $\prec_T$ for Description Logic $\text{ALCNR}$. SIM-DL can be used for our purpose since we use $\text{ALC}$ which is a subset of $\text{ALCNR}$.

Definition 10 ([16]) An exact cover for a CCoP $\forall$ is a solution $(R_c, H_e)$ for $\forall$ such that $T \models H_e \equiv \top$.

5.2. CCoP for Modular Feature Model Configuration

Now, it can be seen that a solution to a concept covering problem corresponds with a configuration for a given modular feature model and a set of stakeholders’ requests. Here,
the aim is to find a cover for the stakeholders’ requests in the context of the modular feature model. An exact cover is equivalent to a feature model configuration that satisfies all of the stakeholders’ requests. However, in cases where such a configuration cannot be found, the best possible configuration is strived for, which will satisfy the maximum number of stakeholder requests. Therefore, it is possible to define the feature model configuration process as a specialization of the concept covering problem:

Definition 11 (Extends Definition 8) Let \( SR_h \) be the stakeholders’ requests, \( F = SKB \cup IC \) be a given modular feature model defined in terms of its structural knowledge and integrity constraints, and \( T \) be a set of axioms defining \( F \), all in a DL \( L \). Let also \( \prec_T \) be an order relation over \( L \) taking into account the TBox \( T \) of \( F \). The modular feature model configuration problem is defined as \( \text{CCoP}_V = \langle L, F, SR_h, T \rangle \). All possible feature model configurations for modular feature model \( F \) based on \( SR_h \) are attained by SOLCCoP(\( V \)).

Following from this correspondence, a solution \( \langle C, H \rangle \) for \( V = \langle L, F, SR_h, T \rangle \) represents a feature model configuration such that:

- \( C \) shows the features of the modular feature model in \( F \) that form the structure of the produced configuration;
- \( H \) consists of all the features that are still remaining in the set of stakeholders’ requests \( SR_h \) that have not been addressed by the configuration provided in \( C \).

So informally stated, a feature model configuration that would satisfy all of the requests of the stakeholders would be a pair \( \langle C, H \rangle \), \( H \equiv \top \). This means that all members of \( SR_h \) are present in \( C \). However, if such a configuration does not exist, a configuration \( \langle C, H \rangle \), where for any other possible derivable configurations such as \( \langle C', H' \rangle \), we have \( H \prec_T H' \), is the best possible achievable configuration, which leaves the least number of unsatisfied stakeholder requests. It is desirable that an exact configuration is sought first, and in case of it being unattainable, a (near) best configuration be developed.

It is well-known that even the basic set covering problem is NP-hard. For concept covering to be of any practical use in a feature model configuration process, it has to be reasonably fast. With this issue in mind, we have extended the tractable polynomial greedy concept covering algorithm proposed for Web Service composition in [10] making it suitable for feature model configuration, building on and extending a classical greedy set covering one [18]. The algorithm takes as input the structural knowledge and integrity constraints of a modular feature model along with a set of specific stakeholder requests, and develops a corresponding configuration.

Algorithm BuildConfiguration(\( F, SR_h, T \))
input concepts \( SR_h, S_i \in F = \cup_j B_j \circ \cup S \), where \( SR_h \) and \( S_i \) are satisfiable in \( T \)
output \( \langle C, H \rangle \)
begin algorithm
/* \( \Delta \) Add all of the implications of the stakeholder requests to \( SR_h \) */
for each \( (f \subseteq \exists \text{hasFeature.} \hat{Y}) \in F \)
    if \( f \in SR_h \) then
SR_h = SR_h \cap \hat{f};
end if
end for each
C = \emptyset;
SR_{hu} = SR_h;
H_{min} = SR_h;
do
S_{MAX} = T;
/* [♣] Perform a greedy search among \( S_i \in F \) */
for each \( S_i \in F \)
  if (C \cup \{S_i\} is a cover for SR_{hu}) and (T \models C \cup \{S_i\} \neq \bot) then
    H = solveCAP(⟨L, S_i, SR_{hu}, T⟩); //solveCAP solves a concept abduction problem.
    /* [◊] Choose \( S_i \) based on an order \( \prec_T \) */
    if H \prec_T H_{min} then
      S_{MAX} = S_i;
      H_{min} = H;
    end if
  end if
end for each
/* [♠] If a new \( S_i \) is found then add \( S_i \) to \( C \) and remove it from \( F \) */
if T \models S_{MAX} \neq T then
  F = F \setminus \{S_i\};
  C = C \cup \{S_i\};
  SR_{hu} = H_{min};
  /* [∞] Enforce modular feature model integrity constraints */
  for each (\exists hasFeature.f \sqsubseteq \exists hasFeature.\hat{f}) \in F
    if f \in C then
      F = F \setminus \{\hat{f}\};
      C = C \cup \{\hat{f}\};
    end if
  end for each
  for each (\exists hasFeature.f \sqsubseteq \neg \exists hasFeature.\hat{f}) \in F
    if f \in C then
      F = F \setminus \{\hat{f}\};
    end if
  end for each
end if
/* [♥] Continue searching until no \( S_i \) is found */
while(S_{MAX} \neq T);
return (C, SR_{hu});
end algorithm

This algorithm attempts to create a configuration that covers SR_h as much as possible, using the concepts \( S_i \in F \). In the first step, stakeholders’ requests are expanded such that they include all of their required child features [\( \triangle \)]. Then, BuildConfiguration adopts a greedy approach towards the selection of the most appropriate features to be included.
in the developed configuration \(\mathcal{C}\), i.e., it solves the concept abduction problem for all the available features in \(\mathcal{F}\), and selects the feature that provides the closest match for \(SR_h\). Once such a feature is identified, it is added to the configuration and removed from the list of stakeholder constraints \(\mathcal{C}\). Furthermore, the addition of this feature to the configuration requires the enforcement of the integrity constraints, i.e., the removal of the features in an excludes constraint and addition of features in an includes relation with the recently selected feature \(\ldots\). This process is repeated until no more appropriate features can be added to the configuration \(\ldots\). Finally, \textit{BuildConfiguration} returns the developed configuration \(\mathcal{C}\). If the configuration does not cover all of the stakeholder requests, i.e., it is not an exact cover, the explanation for why a full cover was not achieved is presented in \(SR_{h,u}\).

6. Case Studies

In this section, we will provide two case studies. The first one builds on the electronic store feature model that we have used throughout the paper to explain step-by-step how a modular feature model can be configured based on our proposed procedure. In this case study, we assume that the monolithic feature model shown in Figure 2 has been modularized and represented as depicted in Figures 4 and 6. In the second case study, we will investigate and compare the performance of the configuration process in terms of the time taken to create a configuration on both the monolithic and modular feature models.

Example 3 Let’s suppose that in the context of the modular feature model discussed in Example 2, the stakeholders have requested a simple configuration of the e-store where the required features are credit card payment, pickup item delivery and a never changing password option. The required features represent the base/minimum set of features that are mandatory (highly desirable) to be present in the final configuration. The stakeholders’ request can be represented as:

\[ SR_h \equiv e - \text{store} \sqcap \text{CreditCard} \sqcap \text{Pickup} \sqcap \text{Never} \]

In order to form a configuration of the modular electronic store feature model, we need to employ the \textit{BuildConfiguration} algorithm. The algorithm evaluates the stakeholders’ requests in several rounds and creates a desirable configuration given the requests. So given \(SR_h\), we show in the following just the first round of the \textit{BuildConfiguration} algorithm is performed. The subsequent rounds of the algorithm are performed similarly. The following steps are marked with the corresponding symbol in the algorithm:

\((\triangle)\) To formulate the configuration, \textit{BuildConfiguration} first adds all of the child features of the requested features in \(SR_h\) to \(SR_h\). As it can be seen, it is only e-store that has constituent features in this example, so the expansion of e-store in \(SR_h\) entails:

\[ SR_h \equiv e - \text{store} \sqcap \text{CreditCard} \sqcap \text{Pickup} \sqcap \text{Never} \sqcap \text{Payment} \sqcap \text{Tax} \sqcap \text{Password} \]

Now, the new features in the expanded \(SR_h\) need to be expanded; hence, password and tax should be unfolded. This process is performed until no further expansion is possible.
The second step consists of a greedy search between the modular feature model features to find the one that provides the most covering over $\mathcal{SR}_h$.

Here, the algorithm chooses a feature that provides the most covering for $\mathcal{SR}_h$ from the features available in the modular feature model. This is calculated based on the ordering created by $\prec_T$. It is easy to see that since the selection of the credit card feature provides a more covering specialization of the feature model for the given stakeholder requests in $\mathcal{SR}_h$, it will be selected.

The feature chosen by the greedy feature selection process, in this case credit card, can be added to the configuration that is being gradually developed. Therefore, $C = \{\text{CreditCard}\}$. Also this feature is removed from the list of available features so that it is not processed in the future rounds of the greedy algorithm.

Once the selected feature is added to the configuration, it must be ensured that its related integrity constraints are also enforced. For this sake, since the credit card feature has two integrity constraints that require the inclusion of the automatic fraud detection and special character passwords features (see Section 2.1); therefore, they will also be added to the configuration. In this step all of the integrity constraints of the features in $C$ are enforced, so now because the automatic fraud detection and special character passwords features are in $C$ their integrity constraints will also be enforced if any.

Finally, the algorithm moves onto the next suitable feature (e.g., pickup) to form the remaining segments of the configuration.

This process is repeated until an acceptable configuration is developed. As it can be seen, if the features initially requested by the stakeholders are consistent, i.e., both them and the features enforced by the integrity constraints can appear simultaneously in a configuration, the BuildConfiguration algorithm is able to create a sound and complete configuration based on the requests (see Section 7 for further details). However, if the requests are not consistent then the algorithm will try its best to satisfy the maximum number of the requests in the developed configuration according to its greedy nature.

Let us now analyze the execution time performance of the modular feature model configuration process in contrast with monolithic feature models.

Example 4 In the second case study, we have studied the impact of modularization on the time required for the configuration (specialization) of a given feature model based on some set of stakeholder requests. To analyze the execution time of the configuration process, we report the results of four scenarios. Each of the scenarios are performed over large-scale synthesized feature models created by the SPLOT Feature Model Generator [19], which is being recently used in the research community for evaluating feature model configuration methods and the execution time evaluations are based on the framework provided by [20]. The scenarios are described as follows:

- The goal of the first scenario was to analyze if modularization enhances the execution time of the configuration process. For this reason, a feature model with 2000 features
Figure 7. Evaluating feature model execution time: a) Scenario 1 on left; b) Scenario 2 on right.

Figure 8. Evaluating feature model execution time: a) Scenario 3 on left; b) Scenario 4 on right.
was used and partitioned into different number of modules. Also the number of stakeholder requests was set to 20. Modularization was performed such that features with more integrity constraints were kept together as much as possible. As it can be seen in Figure 7.a, the configuration process of the feature model with only one module (monolithic feature model) takes much longer than the other cases. This shows that given the same number of features, modularization can enhance execution time of configuration. It should also be noted that although modularization has reduced the execution time, the execution time reduction for modular feature models with more than 6 modules is not significant, e.g. compare the modular feature model with 6 and 9 modules. This is because in this scenario we have kept the number of features at a constant of 2000 and therefore, increasing the number of modules without an increase in the number of features would not enhance the execution time any further. So given this observation, we believe that when the execution time for configuration of the modular feature model remains relatively the same, as is the case for modular feature models with 6, 9, 12, and 15 modules, the decision on how many modules is required is dependant on the semantics of the feature model and how well the features are semantically relevant and how they can be partitioned. The feature model designers would be best for making such design decisions.

In the second scenario, the number of features of the feature model was incremented to see how the increase in the number of features affect the configuration time of different modular feature models. Similar to Scenario 1, the number of stakeholder requests was set to 20. The observations in Figure 7.b corroborate the results of our first scenario. Here, the monolithic feature model (only one module) and the modular feature model with three modules show worse execution time compared with the others. Again, the difference between the execution time for configuration of modular feature models with more modules is quite small, which shows that the designers would be able to select the number of modules of their choice based on the semantic relationships between the features of the feature model.

The other important point to consider is whether the increase in the number of stakeholder requests affects the modularization of a feature model or not. In the previous scenarios, the number of stakeholder requests were set to 20. In this third scenario, the number of features were 2000, but the number of requests increased. As it can be seen in Figure 8.a, the increase in the number of requests negatively affects the monolithic feature model, while the modular feature models stay relatively robust against the increase in the number of requests. This can be in part due to the fact that the requests will be divided between the different modules for satisfaction in the modular feature models, which decreases the execution time for configuration while the requests are all assigned to the one module in the monolithic design that increases the execution time.

In Scenario 4, we have tried to show that modularization of feature model keeps the execution time of configuration relatively stable even if the number of features increase. As shown in Figure 8.b, we have increased the number of modules with the increase in the number of available features, and the configuration time has stayed the same, while for the monolithic feature model, with the increase in the number of features, the execution time grows much higher. This is an indication that modular feature models
are not only useful for enhancing reuse and maintainability but are also effective in enhancing the feature model configuration execution time.

7. Discussions

Modular feature models offer various advantages to the domain analysts, some of which include the possibility of collaborative and distributed development, more efficient feature model evolution, enhanced feature reusability, and others. However, the practicality of the employment of modular feature models depends on the efficiency of the underlying representation formalism that is used to construct such models. Previous experience in representing feature models using Description Logics [8] has shown promising prospect in terms of benefiting from the standard reasoning capabilities of this formal logic. Furthermore, there has been an extensive investigation of the possibility of developing modular Description Logic knowledge bases that provides the opportunity to gain from such developed technologies in the area of feature models. Different modularity frameworks exist that include $E$-connections [21], Interface-based Formalism (IBF) [22], Distributed Description Logic (DDL) [23], and Package-based Description Logics (P-DL) [24]. Each of these formalisms provide some functionality that can be used for different purposes. We believe that DDL can best represent the requirements of a modular feature model. This
is because DDL allows the development of self-contained independent knowledge bases (individual feature modules) that can be interconnected through bridges that connect the modules, which represents in essence the structure of a modular feature model. However, the focus of the other modularity formalisms is not that well aligned with our purpose: 1) in IBF [22] the attention is more on knowledge encapsulation and inheritance, which are not the main concerns in modular feature models; 2) $\mathcal{E}$-connections [21] defines modularity in terms of the connection of modules through connecting rules whose domain is in one ontology and their range in the other, and the basic assumption is that the domain of the modules are disjoint, which is not a condition that we want to enforce for modular feature models; 3) P-DL [24] revolves around the notion of concept import where any module can bring concepts from other modules into its structure. This is in contrast to modular feature models that require feature interaction and mapping which is best supported by the DDL framework.

Furthermore, the Distributed Description Logic framework provides a support suite, called Drago$^3$, that supports visualization and reasoning over modular description logics represented in OWL. So, it is possible to define the feature modules and the sentinel as described before as ontology modules and make use of the visualization and reasoning capabilities of Drago. Figure 9 depicts the visualization of the connection of a feature module with the sentinel in Example 2. The module bridges are shown as lines connecting the feature modules, which makes the comprehension of the formation of the modular feature model easier.

The representation of modular feature models through the employment of the distributed description logic framework also provides the advantage of being able to create a topology of the feature model structure in terms of a (weighted) graph, called the module bridge graph. The topology is developed by representing feature modules as nodes and module bridges as edges. This graph-based representation of modular feature models gives way to a lot of potential techniques from graph theory that can be used to analyze the properties of the modular feature model that can assist in the improvement of the quality of the feature model. This is similar to what is performed in software metrics design [25], for instance the cyclomatic complexity of a program, which is the control flow within that program represented in graph format.

The other important aspect of the proposed work of this paper is the development of feature model configurations from modular feature model descriptions and stakeholder requests. It was shown that a configuration can be developed by making a correspondence with the concept covering problem in DL, a problem similar to set covering. In [18] it has been proven that for a set covering problem, the solution grows logarithmically in the size of the set to be covered with respect to the minimal one. In the $\text{BuildConfiguration}$ algorithm, the complexity source is in the solution of the CAPs and the ordering performed in [♦]. It is possible to show that $\text{BuildConfiguration}$ can be solved in polynomial time. However, it should be noted that the solution provided by this algorithm may not be complete; however, it is always sound. The reason that $\text{BuildConfiguration}$ does not guarantee a complete configuration is due to its greedy nature. Since covering problems are yet to be solved polynomially, and reaching an acceptable solution in a reasonable time is crucial for our problem, we resort to a polynomial greedy algorithm for $\text{BuildConfiguration}$. The algorithm returns a good configuration, which may not always be the best one.
It is worth noting that although the formalism for representing and discussing the modular feature models has been Description Logics in this paper, in reality, feature model modules are developed in Protégé [26], which supports for the manipulation of ontologies in OWL DL. Basically, OWL ontologies are most commonly serialized using RDF/XML syntax, which makes them more easy to understand and work with compared with formal Description Logic representation. For our research, we have been using Protégé and Drago for implementation and testing purposes in terms of creating, manipulating, configuring and visualizing modular feature models introduced in this paper.

Among others, we are interested in further pursuing the work in the paper in two main directions:

1. The tool support for developing modular feature models in the DL formalism is based on available tools such as Protégé for individual feature model design as an OWL ontology and Drago for modular reasoning and visualization. Since it is common practice in the area of feature modeling to create models that conform with the SXFM (Simple XML Feature Model) format and to employ the feature modeling plugin [27] for manipulating feature models, we intend to further develop the modeling plugin to provide support for modular feature model design and configuration. In particular, we are interested to provide developers with an integrated feature modeling environment. In that environment, developers work with multiple feature models and their bridges. Transparently, the feature models are translated into the DDL formalism and reasoning is done. The results of reasoning are then visualized in the integrated environment. This eliminates the need to use many tools in the overall development process;

2. The module bridge graph is an abstract representation of the structure of the modular feature model that can be used to infer interesting properties of the feature model. Recently, there have been some attempts to devise appropriate feature model quality metrics to analyze the structural properties of feature models [28]. These metrics are essentially analogical to those devised as code or object-oriented design metrics [29]. We will focus on the development of such quality metrics based on the module bridge graph, which can include degrees of cohesion, coupling, maintainability, reusability and others.

8. Related Work

Within the realm of software product family engineering, feature models have been viewed as configurable complex systems that can be adapted to produce suitable software products under different circumstances and requirements [30]. Therefore, feature model design is most suitable for complex and large-scale problem domains with numerous features and product options. This requires a structured representation for feature models to be able to manipulate and reason over the developed models. To this end, Mannion was the first to propose the adoption of propositional formula for formally representing software product lines [31]. This idea has been extended by creating a connection between feature models, grammars and propositional formula in [32]. Grammars have also been used in other work such as [3] where context-free grammars have been used to represent the semantics of cardinality-based feature models. This semantic interpretation of a feature model relates
with an interpretation of the sentences recognized as valid by the context-free grammar. However, such techniques can have limitations in defining the integrity constraints of the feature models. For this reason, Czarnecki and Kim employ the Object Constraint Language (OCL) to express such additional restrictions [33]. In their approach, feature models are converted into UML diagrams where integrity constraints are enforced on them through OCL expressions.

Wang et al. have addressed the issue of the formal representation of feature models from a different angle. In their approach, they employ OWL description logic ontologies to model feature model structure and integrity constraints [8]. Their reliance on OWL DL for representing feature models opens up the possibility of employing already existing OWL reasoners to perform feature model configuration validation and verification checks. For instance in their experiments, FaCT++ [34] has been used to see whether a developed configuration is valid or not. Our work in this paper can be seen as a further exploration of the work by these authors. The major advantages of our work is that we provide a modular approach to modeling feature models, and also that we provide an algorithm to automatically build modular feature model configurations based on some intial stakeholder requests, while the work by the mentioned authors is only restricted to monolithic feature models and only perform configuration verification and not configuration development. Along the same lines, the work by Zaid et al. [35] employs description logic ontologies and Semantic Web rules expressed in SWRL [36] to interrelate segmented feature models but only supports for configuration verification.

Similar to the description logic representation where reasoning is performed on feature models using standard DL reasoners, feature model configurations can be verified using Logic Truth Maintenance Systems (LTMS) in their representation in the form of propositional formula [37, 38, 39]. Three of the most widely used methods in this area are Constraint Satisfaction Problem (CSP) solvers [40], propositional SATisfiability problem (SAT) solvers [32, 41], and the Binary Decision Diagrams (BDD) [42]. The basic idea in CSP solvers is to find states (value assignments for variables) where all constraints are satisfied. Although being the most flexible proposal for verifying feature model configurations, they fall short in terms of performance time on medium and large size feature models [43]. Somewhat similar to CSP solvers, SAT solvers attempt to decide whether a given propositional formula is satisfiable or not, that is, a set of logical values can be assigned to its variables in such a way that makes the formula true. SAT solvers are a good option for manipulating feature models since they are becoming more and more efficient despite the NP-completeness of the problem itself [39]. Closely related is the employment of Alloy [44] for analyzing the properties of feature models that is based on satisfiability of first-order logic specifications converted into boolean expressions [45]. Also, BDD is a data structure for representing the possible configuration space of a boolean function, which can be useful for mapping a feature model configuration space. The weakness of BDD is that the data structure size can even grow exponentially in certain cases; however, the low time performance results of BDD solvers usually compensates for their space complexity.

Furthermore, Czarnecki et al. have developed probabilistic feature models where a set of joint probability distributions over the features provide the possibility for defining hard and soft constraints [46]. The joint probability distributions are mined from existing soft-
ware products in the form of Conditional Probability Tables (CPT); therefore, such tables reveal the tendency of the features to be seen together in a software product rather than desirability, i.e., two features may have been seen together in many configurations of a feature model in the past, but they are not desirable for the given product description on hand. Hence, probabilistic feature models are ideal for representing configuration likelihood but not desirability.

Recently, we have proposed the use of the fuzzy extension of \( \mathcal{P}(\mathcal{N}) \), which is standard Propositional Logic extended with concrete domains [41] in order to capture the hard and soft constraints of the stakeholders for being able to effectively reason over the needs of the stakeholders. On this basis, we have developed a maximal covering specialization algorithm that creates a sound and complete specialization of a feature model based on stakeholders hard constraints, which is complemented by the maximal covering configuration algorithm that orders and creates a sound and complete configuration given the soft constraints of the stakeholders. The focus of the techniques developed in this paper is to achieve maximum desirability for the developed feature model configuration for the stakeholders. For this reason, stakeholders’ objectives take center place in the proposed methods. The proposed interactive feature configuration process guarantees the complete satisfaction of the hard constraints of the stakeholders by providing a consistent final configuration for cases where the stakeholders requests are consistent, and provides complete justification for why the hard constraints happened to be unsatisfiable if the requests are inconsistent themselves. \( \mathcal{P}(\mathcal{N}) \) was suitable for dealing with one monolithic feature model where all the facts are known in advance and all features are fully specified. However, when dealing with multiple feature models (i.e., modules), we cannot expect that from the very beginning of the development process all feature model modules will be known and fully specified. Thus, the use of description logic and its open-world assumption supporting incomplete knowledge representation is more suitable for modular feature models.

9. Concluding Remarks

In this paper, we have proposed an approach based on the Distributed Description Logic framework to modularize software product line feature models. The proposed approach allows the independent-distributed development of individual feature modules by different teams of domain experts, and provides for the integration of these individual modules into a unique modular feature modular through the use of module bridges. The modular feature model representation enhances reusability and feature model evolution. Also, in order to develop appropriate modular feature model configurations based on the stakeholders’ requests and requirements, we have created a correspondence between feature model configuration and concept covering in Description Logics. Accordingly, we have developed a greedy algorithm that attempts to find the (near) best possible configuration of the feature model. The models and techniques provide a comprehensive framework for representing and configuring modular feature models in the Description Logic formalism. We have discussed and shown that modularity in feature models increases reusability and maintainability. It also enhances distributed and collaborative feature model design and evolution. Furthermore, our experiments have shown that as feature models become larger and more
complicated, modular feature models tend to stabilize the feature model configuration execution time.

As the two case studies that we reported in the paper demonstrate, the proposed approach can be beneficial for collaboration in cases when teams need to integrate software modules with some overlapping parts or in cases when they are developing new product families where each module is about an independent sub-domain of the studied family. This can then produce many positive implications for different software engineering tasks such as system and business process integration, or leveraging legacy systems in the development of new product families. However, practical implications also open up many research questions to be explored in the future. For example, how can the proposed formalization be used in maintenance of product families. In some cases, a new sub-domain of features (i.e., new modules) needs to be developed and integrated into the existing product family. The question is then whether we need to reconfigure all already deployed product configurations, or it is better to just update them with the new feature modules, which only are to be configured accordingly. Another research question is to investigate relations between commonly used maintenance operators (e.g., update or add) and the proposed DDL-based formalization of modular feature models to automate various feature maintenance tasks.

Notes

1. In this section, ‘feature models’ and ‘modular feature models’ are used interchangeably.
2. We later discuss that developing a non-greedy algorithm is NP-hard.

References