Modularizing Software Product Line Feature Models

Ebrahim Bagheri
Athabasca University and National Research Council Canada
ebagheri@athabascau.ca

Marko Boskovic
Dragan Gasevic
Athabasca University
marko.boskovic@athabascau.ca
dgasevic@acm.org

Faezeh Ensan
University of New Brunswick
faezeh.ensan@unb.ca

Abstract
Feature modeling is a widely used technique for capturing commonality and managing variability in software product lines. In industrial settings, feature models grow very large in size. Furthermore, usually large groups of experts with different capabilities and responsibilities are involved in the design of large-scale feature models. Other than some recent work by researchers, feature models have mostly been treated as a monolithic entity. Treating feature models in such a way makes feature model configuration and management complex and time consuming. In this paper, we introduce the concept of modular feature models. Modularity allows for the distributed development, maintenance and evolution of feature models. We base our modular feature models on the Distributed Description Logics formalism, and propose methods for the configuration and validation of feature models grounded on standard Description Logic inference mechanisms. The results of our experiments show that modularizing feature models reduces the time required for feature model validation and configuration and also enhances the maintainability of feature models.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification/Validation; D.2.13 [Software Engineering]: Reusable Software

General Terms Design, Languages, Theory

Keywords Software product lines, Feature modeling, Modularization, Product line configuration, Model validation, Distributed Description Logics

1. Introduction
The increase in the number of similar requests for software systems, i.e., requests containing extensive common software system requirements has led to the widespread interest in software product line engineering. A software product line (family), is a set of software systems that have many common properties, and share a lot of common reusable assets. Feature modeling is an approach for capturing commonality and managing variability within software product lines [10]. Feature models are usually represented with feature diagrams, a tree-like structure whose root represents the software entity, and its descendant nodes represent the features of that software entity.

In large industrial domains, feature models are hard to manage, debug, and evolve. Feature models are usually developed by a number of experts and teams with different areas of expertise. Typically, various parts of a feature model are developed by different experts and teams, and then these parts are assembled into one comprehensive feature model. The management, debugging and evolution [9] of such large feature models requires the involvement of all experts and teams that participated in the feature model development and is therefore time consuming and costly.

The large size of feature models also causes issues in the feature model configuration process. Feature model configuration is a process that involves the selection of the desired features for a final product. Due to the large size of feature models, manual validation of whether the selected configuration conforms to the constraints specified in the feature model can be time consuming, error-prone, and difficult to perform. Furthermore, due to the large number of features, stakeholders usually specify a limited number of the desired final product features, and expect that the rest of the specialization process is performed automatically.

So given these issues, the contribution of this paper is threefold: 1) The first contribution is the introduction of a formalism for modular feature models. In modular feature models, each module is an individual feature sub-model, referred to as feature model module, which is developed from the perspective of one team of experts. The individual feature modules are assembled into a final comprehensive feature model by specifying the interrelationships between them. Hence, on the one hand, the collection of several feature model modules and their interrelationships form one comprehensive feature model, while on the other hand, each of the individual modules keep their initial structure as originally described by the domain experts, allowing for a modular approach to feature model design. 2) The second contribution is the specification of modular feature models and the verification of the validity of their configurations with Distributed Description Logics (DDL) [30]. DDL allows for loose coupling and autonomy of feature model modules, hence enabling feature module reuse and smoother feature module evolution. Furthermore, DDL allows for the usage of standard Description Logics (DL) [5] inference mechanisms, while improving scalability of reasoning by modularizing a DL representation of feature models. 3) The final contribution is an algorithm for the interactive modular feature model configuration. Given a certain modular feature model and several stakeholder requests, the algorithm will compute a feasible configuration from the modular feature model. The algorithm is essentially based on the Concept Covering Problem (CCoP) [15] in DL. It considers the stakeholders’ requests as one concept and tries to find a feature model configuration that covers the highest possible number of requested features.
In brief, we show how our modular approach to feature modeling provides means to cater benefits such as distributed and collaborative feature model development, reuse of feature modeling artifacts, faster and more efficient feature model configuration and also easier evolution and management of feature models. We will show how our proposed approach is novel compared to existing feature modeling formalisms, i.e., it provides a well rounded solution for developing modular feature models: supporting both the representation and modeling of modular feature models as well as providing supportive tools and methods for interactively configuring and validating modular feature models. So, the strong point and novelty of our work is that it provides both the required formalisms for representation and the needed tools for configuration of modular feature models. These are two features which are not simultaneously present in the work by other researchers.

This paper is structured as follows. Section 2 introduces modular feature models. The specification of modular feature models in DDL, validation with DL inference mechanisms and automated configuration based on CCoP are depicted in Section 3. The evaluation of the validation and configuration routines and their response times are described in Section 4. Section 5 discusses the related work. Finally, Section 6 gives some final remarks and outlines the future work.

2. Modular Feature Modeling

A feature model is a hierarchical structure of features, classified as mandatory, optional, alternative feature group, and or feature group, and graphically represented as in Figure 1.a). Mandatory features must be included in the description of its parent feature, optional may or may not, from an alternative feature group exactly one can be included and from an or feature group at least one.

Beside the parent-children relationships, in feature models also integrity constraints exist. The most important are includes and excludes integrity constraints. They specify that the presence of a given feature (set of features) in a feature model configuration requires the existence of another feature (set of features) in that same feature model configuration, respectively. More details can be found in [13].

Other than some recent developments [3, 18, 27, 28], feature models have mostly been considered as monolithic entities. This work introduces modularity in feature modeling, an engineering technique that builds larger systems by combining smaller components. In feature models, this translates into composing a large feature model from independent self-contained modules. An example of a modularization of the feature model in Figure 1.b has been depicted in Figure 1.c.

The modular feature model in Figure 1.c consists of four modules. Each module is a viewpoint from different perspectives on the whole comprehensive feature model. Feature Module 1 is a security viewpoint. It shows the security mechanisms embedded in the product family. A human resources perspective is given in Feature Module 2, i.e., it shows all features of the e-store that require the involvement of the e-store employees. Feature Module 3 shows all features of interest to the financial department. Finally, all features related to transport are contained in Module 4.

An alternative approach to the modularization of the feature model in Figure 1.b is presented in Figure 1.d. Modules of this modularized feature model are developed by four teams: teams of experts in payment systems, financial tax experts, transport experts, and computer security experts, respectively. Given these two different ways of modularizing the monolithic feature model in Figure 1.b (one approach puts functionally similar features in the same modules, and the other approach, groups the features present in a given target application in similar modules), it is possible to say that similar to the modularization of a software program, feature models can and need to be modularized and hence a modularization formalism is required to support this process. Assuming that feature modules need to be independent self-contained sub-feature models themselves, they can be interrelated by inter-module integrity constraints in order to form a larger feature model as a result of their integration. Here, the main characteristic of the inter-module integrity constraints is that they relate features of different feature modules together, i.e., two feature modules can be integrated by specifying how some of their internal features align with each other and are hence related. In contrast to inter-module integrity constraints, each feature module can have its own internal integrity constraints (like any other self-contained feature model), which are referred to as intra-module integrity constraints.

Generally, three inter-module integrity constraints can be defined: inclusion, exclusion, and equivalence. An inter-module inclusion constraint specifies that the presence of a given feature (set of features) of a given module in the feature model configuration requires the existence of another feature (set of features) of another module in that feature model configuration. In Figure 1.d, one can define inclusion integrity constraints between Cash and Pickup, E-delivery and Credit Card, and Credit Card and Special Char as examples for inter-module inclusion integrity constraints between Feature Modules 1 and 3, 3 and 1, and 1 and 4, respectively. Also, inclusion integrity constraint between Credit Card and Automatic, and Never and Special Char can be defined as intra-module inclusion integrity constraints for Feature Modules 1 and 4, respectively. This is because these integrity constraints only refer to the features from the same module and do not work between multiple modules. Similarly, the exclusion inter-module integrity constraints are used to specify that the presence of a given feature (set of features) of a given feature module in a feature model configuration requires the elimination of another feature (set of features) of another module from that feature model configuration. In Figure 1.d), exclusion integrity constraint between Pickup and Universal Tax Rate can be an inter-module integrity constraint between Feature Modules 3 and 2. An exclusion integrity constraint between Manual and Credit Card features is an intra-module integrity constraint of the Feature Module 1. Finally, an equivalence inter-module integrity constraint between two features of different modules can be derived as a pair of oppositely directed inter-module inclusion integrity constraints. For example, the Manual feature of Feature Module 1 in Figure 1.c is equivalent to the Manual feature of Feature Module 2.

3. The Modularity Formalism

 Distributed Description Logics (DDL) [30] is a formalism that allows for the loose coupling of DL knowledge bases [5]. Such a formalism improves the modularity of knowledge bases, enhances the ability of knowledge bases to evolve, and facilitates scalable reasoning. The reason why DDL has been chosen as opposed to our own DL modularity formalism, IBF [16], or others such as PDL or E-connections has been discussed in [6]. In this section, we demonstrate how DDL can be used for the software product line specification, configuration validation, and semi-automatic feature model configuration.

3.1 Modular Feature Models in DDL

Wang et al [32] have already shown how the structure of a feature model and the stakeholders’ requests can be represented using Description Logics. Readers not familiar with basics of Description Logics are encouraged to browse Chapter 1 of [21]. Here, we just provide some short examples as representatives and encourage the interested reader to see [32] for more details. As an example, the Method feature and the Credit Card, Debit Card, and Cash feat-
Figure 1. Feature diagrams: a) graphical representation of different feature types, b) a monolithic feature model, c) modular feature model for security, human resources, financial, and transport domains, d) A modular feature model for payment, tax, shipping, and password domains.

Features can be specified as:

\[
\text{Method} \sqsubseteq (∃ \text{hasFeature.CreditCard}) \sqcup (∃ \text{hasFeature.DebitCard}) \sqcup (∃ \text{hasFeature.Cash})
\]

Similarly, the exclusion integrity constraint between Manual and Credit Card features can be defined as:

\[
∃ \text{hasFeature.Manual} \sqsubseteq ¬(∃ \text{hasFeature.CreditCard})
\]

Given the description of feature models and stakeholder requests in DLs, we will assume the following knowledge bases in the rest of the paper: SKB - The structural knowledge of a feature model is represented using DL concepts and roles; IC - Integrity constraints are subsumption relations; SRh - Stakeholders’ requests are complex concept descriptions.

Now that the definition of monolithic feature models is feasible in DL, we will assume that each modular feature model is a monolithic feature model by itself, which can be interrelated with others such that a modular feature model is formed. In order to interrelate different monolithic feature models with each other, different integrity constraints can be used to show how the various features of each feature model are related. These will form the inter-module integrity constraints. For instance, in order to integrate the shipping and payment feature modules with each other shown in Figure 1.c, one can express an inter-module integrity constraint such as Airmail includes Credit Card. In this way, the two feature modules become connected and form a larger modular feature model. We formally define our modular feature model formalism as follows:

**Definition 1.** Given a set \( F = \{F_i\}_{i∈I} \) of feature model modules, 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 \( T = \{T_i\}_{i∈I} \).

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 model 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 inter-module integrity constraints.

**Definition 2.** Given a family of feature model modules and their corresponding TBoxes \( T = \{T_i\}_{i∈I} \), an inter-module integrity constraints from module \( i \) to \( j \) is a feature model integrity constraint of one of the following forms: \( (i : X \sqsubseteq Y) \) and \( (i : X \sqsubseteq Y) \), 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 inter-module integrity constraints can be derived as a pair of inter-module integrity constraints, denoted \( i : X \rightarrow S j : Y \) (\( i : X \rightarrow Y \) and \( i : X \rightarrow S j : Y \)) to show that the features refer to the same concept.

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

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

Also, an equivalence relationship is used when the same feature is introduced in different modules, be them with the same name or under dissimilar names, and therefore they would need to be shown to be the same feature so that when reasoning is performed they would be considered to be the same entity.

Feature modules can be integrated through the application of integrity constraints. Simply stated, inter-module integrity constraints 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 the equivalence relationship.

Given the notion of inter-module integrity constraints, it is possible to define modular feature models as a collection of feature modules and their interrelating inter-module integrity constraints. 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:

\[
\begin{align*}
\text{GenericFeature} &\subseteq \top \\
\text{hasGenericFeature} &\subseteq \text{ObjectProperty} \\
\text{GenericProductLine} &\subseteq \exists \text{hasGenericFeature}.\text{GenericFeature} \\
\text{OurProductLine} &\subseteq \text{GenericProductLine}
\end{align*}
\]

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, \( \mathcal{S} \mathcal{R}_a \), 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 integrated 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 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 address a totally different problem. In essence, the 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 \( (\mathcal{F}, \mathcal{B}, S) \) where \( \mathcal{F} = \{ \mathcal{F}_i \}_{i \in I} \) represents the individual feature modules, \( \mathcal{B} = \{ \mathcal{B}_i \}_{i \in J} \) denotes the set of all available inter-module integrity constraints, and \( S \) is the feature model sentinel.

**Example 2.** Let us consider the feature modules depicted in Figure 1.d. A modular feature model developed from these four individual feature modules consists of a feature model sentinel \( S \), four feature modules \( \mathcal{F}_i \), \( i = 1..4 \), and several inter-module integrity constraints including those described in Example 1.

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 inter-module integrity constraints, denoted as the set \( \mathcal{B}_S \). Each of the inter-module integrity constraints between the sentinel and the other feature modules \( (\mathcal{B}_S) \) consists of the following axioms:

\[
i : \text{Feature} \rightarrow S : \text{GenericFeature}
\]

Also the sentinel is given access to the top feature(s) of each module through an inter-module integrity constraint \( (\mathcal{B}_S) \) in order for us to be able to build the overall structure of the modular feature model in the sentinel. For instance, an inter-module integrity constraint such as \( \mathcal{B}_S \) can consist of \( (S : \text{Payment} \rightarrow 1 : \text{Payment}) \), 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 inter-module integrity constraints, a complete feature model can be put together in the sentinel for e-store as follows:

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

**3.2 Feature Model Configuration Validation**

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 DL inference mechanisms to perform the configuration validation process. Any DL system provides two basic standard reasoning mechanisms:
Concept satisfiability $T \models D \sqsubseteq \perp$. 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 $T \models D \sqsubseteq C$. 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 DL 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 $T \models PL \cap C \subseteq \perp$, which is called the satisfiability of concept conjunction; 3) the feature model configuration is not at all compatible with the feature model $T \models PL \cap C \not\subseteq \perp$, referred to as the entailment of concept disjointness [17]. 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 DL Tbox, we use the DDL bridge operator to make the modular feature model ready for reasoning using those standard reasoning mechanisms.

Definition 5. Given a set of inter-module integrity constraints $B_{ij}$ 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 [30]. For our purpose, we need to perform the bridge operator on and from features involved in the configuration onto the feature model sentinel. In this way, the sentinel will temporarily contain the representative 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: $\cup_i B_{ij}(\cdot) \cup S \models C \subseteq PL$, where $B_{iS}(\cdot)$ denotes the application of the bridge operator from the involved features onto the sentinel, and $i$ iterates over the participating feature modules. With this standard DL reasoning mechanism, designers can easily validate their feature model configurations.

One of the most interesting aspects of the transformation process is that it is only a partial import into the sentinel. This means that once a feature model is put together as a result of the integration of multiple feature modules through the sentinel, and a reasoning process needs to be performed on the newly integrated feature model, only those features that are involved in the development of this new feature model will be included temporarily in the sentinel. As we will see in our experiments shown in the Evaluation section, the implication of this is that the time required for feature model configuration and configuration validation will be significantly reduced. According to the experiments performed by White et al. [33], as the number of features within the feature model increases the required diagnosis time also increases. This implies that the configuration and configuration validation time of a feature model is dependent on the size of the feature model. So, it is clear that when a feature model is created by putting multiple feature modules together, the more feature modules are put together the higher the processing time will be. However, in our approach in order to address this issue, we use the bridge operator. The use of the bridge operator and the transformation into the sentinel reduces the time required for the configuration and configuration validation of a feature model for two reasons: 1) only those modules that have been included in the formation of the new feature model will be included in the transformation process. This readily reduces the number of features that are imported by not including irrelevant feature modules; however, in standard techniques with a monolithic approach, all of the feature space is considered by the configuration and validation algorithms even the irrelevant features are considered; 2) the bridge operator will only import those features of the involved feature modules into the sentinel that have some form of compulsory involvement in the new feature model, i.e. they are mandatory features, features that have been requested by the stakeholders, features that are in and relationship with a compulsory feature or similar cases. For these two reasons, the transformation performed through the bridge operator minimizes the involvement of unnecessary features within the configuration and validation processes and hence reduces the required time as shown later in the Evaluation section.

To put things in the context of the software product line development life cycle, it should be noted that the development of the individual feature modules is a process that is performed in the domain engineering phase, i.e., the scope of each feature module is defined and the features that need to be incorporated in each module are identified. Then, the required features are placed and represented in their corresponding feature modules and the integrity constraints between these features in each module are specified. As opposed to the monolithic form of feature models, modular feature models benefit from two conceptual forms of integrity constraints. Syntactically these integrity constraints are very similar, though they are conceptually different. We refer to these two types of integrity constraints as application-independent and application-specific integrity constraints (Application-independent and application-specific constraints are defined using the inter or intra-module integrity constraints. Essentially if an inter or intra-module integrity constraint is related to a given application we refer to it as application-specific and if it is a general purpose constraint that should hold in all applications of the domain we call it application-independent). Application independent constraints should always hold regardless of the target application where they have been defined. On the other hand, application-specific constraints are those that are not necessarily valid for all applications but should hold within a specific application; therefore, in the domain engineering phase, we only define the application-independent integrity constraints. The artifacts developed as the outcome of the domain engineering phase are the individual feature modules and their application-independent integrity constraints.

Now, in the application engineering phase, the sentinel is created and the application-specific integrity constraints between the individual feature modules are formulated based on the target application. In this phase, the requirements of the stakeholders are considered and a valid feature model configuration is formulated. So, as it can be seen, since the development of individual feature modules is performed in the domain engineering phase, they are essentially independent of each other and are only interrelated in the application engineering phase through the use of the sentinel (which acts like a glue to put them together) and the employment of application-specific integrity constraints. Given this characteristics of the feature modules (their independence), they can be used in different configurations to form a target application of interest in the application engineering phase.

In the following section, we will benefit from and customize a method originally proposed by Ragone et al. for distributed Web service composition [26]. In their method, concept abduction is used to address the concept covering problem in DL, 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.

3.3 Modular Feature Model Configuration

In practice, stakeholders rarely specify all of the desired features that cover the complete product line configuration space. Instead, they rather specify a set of desired features, and expect the whole system to be specialized accordingly. For this reason, in this section we provide an algorithm that takes as an input a set of stakeholders’ features and returns a specialization which covers the highest possible set of stakeholders’ requests in $S_R$. 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 $S_R$. Based on this last remark, we use solutions to sets of CAP to create a correspondence between modular feature model configuration and CCoP. We provide the following definition for the concept covering problem.

**Definition 7.** 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 a DL $L$, where $D$ and $S_1, \ldots, S_k$ are satisfiable in $T$. Let also $\prec_T$ be an order relation over $L$ taking into account the TBox $T$. The Concept Covering Problem (CCoP) denoted as $\forall=(L, D, T)$ is finding a pair $(R_c, H)$ such that

1. $R_c \subseteq R$ and $T \not\models \bigwedge_{S_i \in R_c} S_i \sqsubseteq \bot$;
2. $H \in \text{SOL}(L, D, T)$ and $H \prec_T D$.

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

In the above definition for the solution $(R_c, H)$ of a CCoP, $R_c$ shows which concepts in $R$ represent the cover for $D$ w.r.t. $T$ and $H$ depicts 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. $T$, and the abduced concept $H$ represents what is still in $D$ and is not covered by $R_c$.

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 an exact cover exists, it is independent of any order $\prec_T$ on the DL adopted. For our purpose, ordering $\prec_T$ can be achieved with the $\text{Sim-DL}$ algorithm presented in [22]. The $\text{Sim}$-

---

**Figure 2.** The proposed BuildConfiguration algorithm.

Algorithm $\text{BuildConfiguration}(F, S_R, T)$

input concepts $S_R$, $S_i \in F = \cup_i B_i; \cap_i S_i$, where $S_R$ and $S_i$ are satisfiable in $T$.

output $(C, H)$

begin algorithm

/* [1] Add all of the implications of the stakeholder requests to $S_R$, */

for each $(f \sqsubseteq \exists hasFeature ) \in F$

if $f \in S_R$

$S_R = S_R \cap f$

end if

end for each

$C = \emptyset$;

$S_{R_{h\_n}} = S_R$;

$H_{\min} = S_{R_{h\_n}}$;

do

$S_{MAX} = T$;

/* [2] Perform a greedy search among $S_i \in F$ */

for each $S_i \in F$

if $(C \cup \{S_i\})$ is a cover for $S_{R_{h\_n}}$ and $(T \not\models C \cup \{S_i\} \sqsubseteq \bot)$

$H = \text{solveCAP}(\langle L, S_i, S_{R_{h\_n}}, T \rangle)$;

/* [3] Choose $S_i$ based on an order $\prec_T$ */

if $H \prec_T H_{\min}$

$S_{MAX} = S_i$;

$H_{\min} = H$;

end if

end if

end for each

end while

/* [6] Add a new solution if it is found */

if $T \not\models S_{MAX} \neq T$

$F = F \cup \{S_i\}$;

$C = C \cup \{S_i\}$;

$S_{R_{h\_n}} = H_{\min}$;

/* [6] Enforce modular feature model integrity constraints */

for each $(\exists hasFeature, f \sqsubseteq \exists hasFeature ) \in F$

if $f \in C$

$F = F \backslash \{f\}$;

$C = C \cup \{f\}$;

end if

end for each

end while

end algorithm
DL algorithm computes the degree of overlapping of two concepts. Herewith, it is possible to create the ordering of a set of concepts according to their degree of overlapping with previously selected external concepts. Sim-DL is developed for $\text{ALCNR}$, which is expressive-enough for our feature modeling formalism.

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 configuration cannot be found, the best possible configuration is strived for, which satisfies the maximum number of stakeholder requests. Hence, it is possible to define the feature model configuration process as a specialization of the concept covering problem.

Informally stated, a feature model configuration that would satisfy all of the requests of the stakeholders would be a pair $\langle C, H \rangle$, where $H \equiv \top$. This means that all of members of $\text{SR}_{\text{hi}}$ 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 $H' \not<_{\text{T}} H^*$, 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 devised a tractable polynomial greedy concept covering algorithm for feature model configuration, building on and extending a classical greedy set covering one [12]. The algorithm takes as input the structural knowledge and integrity constraints of a modular feature model along with a set of stakeholder requests, and develops a corresponding configuration.

The above algorithm shown in Figure 2 attempts to create a configuration that covers $\text{SR}_{\text{hi}}$ using the concepts $S_i \in \mathcal{F}$. In the first step, stakeholders’ requests are expanded such that they include all of their required child features $\lnot \Delta$. Then, BuildConfiguration adopts a greedy approach towards the selection of the most appropriate features to be included in the developed configuration $\lnot \mathbf{S}$, i.e., it solves the concept abstraction problem for all the available features in $\mathcal{F}$, and selects the feature that provides the closest match for $\text{SR}_{\text{hi}}$. Once such a feature is identified, it is added to the configuration and removed from the list of stakeholder constraints $\lnot \mathbf{S}$. 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. Here we assume that features higher in the tree are analyzed before their children $\lnot \mathbf{X}$. This process is repeated until no more appropriate features can be added to the configuration $\lnot \mathbf{S}$. Finally, BuildConfiguration returns the developed configuration $C$. If the configuration does not cover all of the requests, i.e., it is not an exact cover, the explanation for why a full cover was not achieved is presented in $\text{SR}_{\text{hi}}$. The description of the algorithm in more detail on a real world case study can be found in [6].

4. Evaluation

Let us now analyze the execution time performance of the modular feature model configuration process in contrast with monolithic feature models. In this study, we have evaluated 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 [25], 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 [11]. 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 was used and partitioned into different number of modules. Also the number of stakeholder requests was set to 20. As it can be seen in Figure 3.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 dependent on the semantics of the feature model and how well the features are semantically relevant and how they can be partitioned. The feature model designer would be best for making such design decisions.

In the second scenario, the number of features was increased 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 3.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 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 3.c, 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 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 3.d, 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.

The reason why the use of the modular feature models reduces the execution time were discussed earlier in Section 3.2. The main
Explanation for it is that the configuration and validation of a feature model developed from a set of interconnected feature modules will only consider and include the set of relevant features of the involved modules for the new feature model and will not even process the irrelevant features in the other modules; therefore, the execution time is reduced. However, in cases where all features are stored in a single feature module (a monolithic feature model), all of the features, be them relevant to the configuration or not, will be processed, which as shown by White et al. [33] increases the execution time.

The results of these experiments are an indication that our formulation of modular feature models are both useful for enhancing reuse and maintainability in feature models and are also effective in reducing the feature model configuration execution time.

5. Related Work

The organization, validation, and assistance in configuration of large feature models has been a topic of research in academia for quite sometime. For instance, an approach for the organization of feature models in several interconnected feature sub-models has been introduced by Czarnecki et al. [13]. In their work, each feature sub-model corresponds to one stage in the staged configuration process. Each stage has its own feature model and a set of stakeholders responsible for configuring the feature model. However, depending on the selection of previous stages, the stakeholder receives a specialized feature model, i.e., the feature model from which some certain features are removed in order not to have conflicts with previous stages. This approach can be inconvenient for realistic scenarios, where the stakeholders of one stage do not configure all features of their domain, but rather only subset of their initially required features is selected. In the configuration method suggested by Czarnecki et al. [13], some automatically selected features of previous stage, may remove an important features needed to be specified at later stages. For this reason, instead of specifying feature model management operations, we specify logic relation between features, and use these relations for identifying conflicts between stakeholders. The final resolution of these conflicts is done in a collaborative consulting process.

The application of logical formalisms in feature models has also been widely used. Mannion [23] was the first to propose the adoption of propositional formula for formally representing software product lines. Although that work introduced the validation of product line models and configurations, it did not provide means for guiding developers to specify syntactically correct feature models. This has been resolved by Batory [7] who suggested specifying feature models as grammars, and transforming grammars into first order logic. Additionally, as means for detecting inconsistencies in configurations Batory [7] suggests usage of Logic Truth Maintenance Systems (LTMSs). LTMSs completely resolve conflicts in configurations.

Alternatively, we employ DL for the detection of inconsistencies in feature model configurations. DL completely resolves inconsistencies in feature models with detection of those inconsistencies in their DL TBox representations. Additionally, by representing configurations also as TBox specifications, and by employing DL standard reasoning mechanisms, it is also possible to detect inconsistencies in configurations. To this time, Zaid et al. [34] have applied OWL DL ontologies and Semantic Web rules to interrelate segmented feature models but their work only supports for configuration validation. Wang et al. [32] have also addressed the issue of the formal representation and validation of feature models with DL, only from a different angle from Zaid et al. [34]. In their approach, they employ OWL DL ontologies to model feature model structure and integrity constraints. Their reliance on OWL DL for representing feature models opens up the possibility of using existing DL reasoners for feature model configuration validation. However, they do not provide support for modularity.

In order not only to verify, but also assist in configuration by finding states where all constraints are satisfied, several approaches have been proposed. A very comprehensive and useful review of techniques in the area of automated analysis of feature models has
been published by Benavides et al. [8]. We will not review these techniques since they are not directly relevant to our work.

There has recently been some work which has tried to address the issue of feature model evolution and maintenance. For instance, the viewpoint-oriented feature variability modeling framework [24] uses the concept of viewpoint, which is often used in the requirement engineering domain [31] to form independent but interrelated feature models. This work only confines itself to defining specific rules for handling conflicts between the information contained in each of the individual feature models related to the viewpoints. Another approach to dealing with this issue has resulted in the development of supplier-independent feature models [20]. In that approach, each supplier provides a version of its feature model which is referred to as the Supplier-Specific Feature Model (SSFMs). SSFMs will then be integrated into a higher level feature model that provides support for supplier variability and alongside with the supplier-independent feature model form what is called the composite-supplier feature model. Although the idea of supporting supplier-specific feature models is quite interesting, it lacks scalability and is too complex to be suitable for developing easy reasoning support. In addition, it is not clear how evolution is handled with the context of the supplier-independent and supplier-specific feature models. In contrast, our modularity formalism has been designed with evolution in mind such that feature modules can evolve independently of the other feature modules making the support for maintenance and evolution of feature models quite easy. Also, on the contrary to the viewpoint-oriented feature variability modeling framework, our formalism does not require the complex integration of feature modules; therefore, reasoning is facilitated.

Similar idea to [20], on composing one monolithic feature model from several separated feature models is presented by Acher et al. [2, 3]. The authors introduce Multiple Software Product Lines (MPLs) and the concept of composing feature models. An MPL consists of a set of SPLs which share common features. Feature models of each SPL in an MPL is developed independently. Feature models of independent SPLs are, with the usage of a merging operator, automatically merged into one monolithic feature model of the MPL. While the approach of Acher et al. creates one monolithic feature model that represents the whole MPL, our approach leaves the combined feature model modularized and, so, reduces the time and required resources for configuration verification. Additionally, our approach enables specification of inclusion, exclusion, and equivalence inter-module constraints, while the approach of Acher et al. uses shared features for merging; therefore, supporting only equivalence inter-module constraints. By facilitating a larger number of inter-module constraints, our approach puts more effort in integration of feature models. Also, our approach enhances separation of concern, because it enables integration of separate feature modules that do not necessarily have common features.

There has also been focus on reducing the maintenance effort for software product line feature models by Deepak Dhungana et al [14]. In their work, software product lines are organized as a set of interrelated model fragments, each of which defines the variability of a specific aspect of the system. The authors of [14] provide means for the semi-automatic merging of the product line fragments and also for the automatic detection of inconsistencies that may exist among the different fragments.

Hartmann and Trew have promoted the idea of context variability models for representing multiple software product lines [19]. Their model constrains the feature model, which makes the process of dealing with multiple feature models for supporting different contexts and dimensions possible [1]. The work is based on an extended form of feature models referred to as the Multiple Product Line feature model (MPL-feature model). The other quite relevant work is the use of multi-level feature trees for hierarchically organiz-

6. Conclusions and Future Work

In this paper, we have proposed an approach based on the Distributed Description Logic framework to modularize, configure and reason about software product line feature models. In a nutshell, our proposed work provides the following advantages to the feature modeling practice: 1) While some researchers have provided means for partitioning feature models into smaller domains that are interconnected, no formal technique for reasoning over such feature models for creating model configurations or validating an existing configuration has been proposed. However, we propose a well rounded work that both provides formalisms for representing and modeling modular feature models and also means for configuring and validating them; 2) We differentiate between application-specific and application-independent integrity constraints in that application-specific constraints can be formulated to bring several modules together in order to form a new feature model for a given application. These constraints may not hold in a general setting but are correct for a given application. On the other hand, application-independent constraints are general feature model integrity constraints that need to always hold. 3) The formalization provided for the representation of feature models in our work allows for the modeling of flat, nested and hierarchical feature modules through the inter-module integrity constraints, each of which can be useful in different situations; 4) The configuration and validation algorithms that have been proposed in this paper are based on well-established DL reasoning mechanisms and have already been used for and validated in Web service composition [26].

As the experiments that we reported in the paper demonstrate, the proposed approach can be beneficial for collaboration in cases when teams need to integrate 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.

In our future work, we are going to focus on the enhancement of this methodology in two dimensions. While this paper introduces verification and semi-automatic configuration of so called hard requirements, we intend to extend the approach to soft requirements, i.e., requirements such as performance, or reliability at some certain level. The second direction in which we intend to work is the evaluation of the quality of feature models such as cohesion, coupling, and reusability. We intend to develop formal models for analyzing and suggesting the improvement of these quality metrics. We
see modular feature models as a promising approach for separating work between experts in different domains. As such, we consider improvements in previously mentioned directions necessary for enhancing the development, configuration and maintenance of feature models. Currently, we have a prototype toolset support for creating, configuring and verifying modular feature models based on the DDL Drago toolkit (For the description of the toolset, see [6]). We intend to extend our toolset to interact with the Feature Modeling plugin developed by Generative Programming Lab at the UWaterloo [4]. Sergio Segura et al. have very recently developed an approach for test data generation for testing feature models [29]. We are also interested in analyzing the performance of our approach with the use of the test data generated using their approach.

References


Some Notes For Respected Reviewers

We would like to clarify the difference between the contributions of this paper and the other papers that we have published recently in SPLC 2010 and also another paper that we have also submitted to GPCE 2010:

In Paper 1, we have proposed a framework for dealing with soft constraints of the stakeholders. The approach that we have in that paper is to use fuzzy linguistic variables for representing soft constraints, i.e., quality attributes. Then we formulate a sound and complete algorithm for configuring feature models annotated with soft constraints using the P(N) language and the MAX-WEIGHTED-SAT algorithm.

In Paper 2, we have extended the Analytic Hierarchy Process in order to choose the best software features from among the available features of a feature model. The selection of the best features in our approach, called S-AHP, allows the application engineers to provide the feature model configuration processes with the most suitable features and also develop a better customized target application.

In Paper 3, we have proposed to interrelate between requirements engineering goal models and software product line feature models in order to create a bridge between the stakeholders intention space and the available feature space. This bridge can help the stakeholders to decide which features of the feature model are most suitable and most relevant for the needs and objectives that they have. We have used the forward and backward label propagation algorithms to facilitate this process.

As it can be seen the theme of these three papers are essentially different from this paper that addresses modularity in software product line feature models.

