Modular Feature Models: Representation and Configuration

Ebrahim Bagheri
School of Computing and Information Systems, Athabasca University, 1 University Drive, Athabasca, AB T9S 3A3, Canada
Electrical and Computer Engineering Department, University of British Columbia, Forest Sciences Centre
Building, FSC 2623–2424 Main Mall, Vancouver, BC V6T 1Z4, Canada
ebagheri@athabascau.ca

Faezeh Ensan
Sauder School of Business, University of British Columbia, 2053 Main Mall, Vancouver, BC V6T 1Z2 Canada
faezeh.ensan@sauder.ubc.ca

Dragan Gašević and Marko Boškovic
School of Computing and Information Systems, Athabasca University, 1 University Drive, Athabasca, AB T9S 3A3, Canada
School of Interactive Arts and Technology, Simon Fraser University Surrey, 13450 102 Ave., Surrey, BC V3T 5X3, Canada
{dgasevic, marko.boskovic}@athabascau.ca

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.

ACM Classification: D.2.4, D.2.13

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 (Pohl, Bockle and Van Der Linden, 2005). 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 (Kang, Cohen, Hess, Novak, Peterson and INST, 1990; Czarnecki and Eisenecker, 2000). Feature modeling is an important conceptual tool that offers modeling support for software product
Modular Feature Models: Representation and Configuration

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 90s by the Software Engineering Institute. Its concepts were developed within the Feature Oriented Domain Analysis (FODA) methodology (Kang et al., 1990). 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) (Simos, 1995), FeatuRSEB (Griss, Favaro and d'Alessandro, 1998), Feature-Oriented Reuse Method (FORM) (Kang, Kim, Lee, Kim, Shin and Huh, 1998) and cardinality-based feature modeling (Czarnecki, Helsen and Eisenecker, 2005).

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 (2003) 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) (Regio and Greenfield, 2005) 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 dedicated to particular subdomains of the SPL (Macala, Stuckey and Gross, 1996; Dikel, Kane, Ornburn, Loftus and Wilson, 1997). 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. 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 (Czarnecki, Helsen and Eisenecker, 2004), where outcome of each stage is a feature model specialization.

Having previously stated in mind, in this paper, we are specifically looking at addressing the following problem statements:
Modular Feature Models: Representation and Configuration

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;

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 dependencies 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.

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. In order to tackle the two mentioned challenges, we take a holistic approach where both problems are viewed in a unified framework. Fundamentally, modularity and concept covering lay at the heart of our framework. 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 extendibility, reusability, parallel and collaborative design progression and others. Concept covering is a computational minimization process that evaluates whether a certain conceptual structure wraps around another and can be used to automatically develop configuration based on the feature model constraints and the stakeholder requirements. Furthermore, it 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. We adopt Description Logics (Baader, Calvanese, McGuinness, Patel-Schneider and Nardi, 2003) as the underlying formalism for representing feature models, and in particular Distributed Description Logics (Borgida and Serafini, 2003) which enables the introduction of modularity in such a representation. Furthermore, representation in Description Logics enables use of already developed algorithms for concept covering in models specified in Description Logics (Ragone, Noia, Sciascio, Donini, Colucci and Colasuonno, 2007).

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 5. In Section 6, two case studies are presented. Section 7 provides discussion on the proposed solution. 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 (Kang, Lee and Donohoe, 2002a). They are widely used for depicting the shared structure and behaviour 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
Modular Feature Models: Representation and Configuration

Modular Feature Models are typically represented using a tree structure, where each node represents a feature and the root node represents the domain concept. In this context, features are classified into:

- **Mandatory**: The feature must be included in the description of its parent feature (e.g., Payment, Tax, and Password).
- **Optional**: The feature may or may not be included in its parent description (e.g., Shipping).
- **Alternative feature group**: One and only one of the features from the feature group can be included in the parent description (e.g., children of the Tax feature).
- **Or feature group**: One or more features from the feature group can be included in the description of the parent feature (e.g., children of the Shipping feature).

The tree structure of feature diagrams is limited in representing the complete set of mutual interdependencies. To address this, additional constraints are added to feature models, known 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).

![Feature Model Diagram]

Figure 1: The online electronic store feature model.
Excludes, the presence of a given feature (set of features) requires the elimination of another feature (set of features).

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 includes credit card payment; 6) credit card payment includes special character passwords; 7) pickup excludes universal tax rate.

In addition to this information, several feature modeling notations have additional constructs, such as the cardinality-based feature models (Czarnecki et al., 2005) that provide means for expressing the plausible cardinality of the features, and notation introduced by Kang, Lee and Donohoe (2002b) that add relationships like ‘composed-of’, ‘generalization’, and ‘implemented-by’ (Kang et al., 2002b). However, most of the approaches consider only the configuration semantics (Schobbens, Heymans and Trigaux, 2006), which is also in the scope of our work.

3. FEATURE MODEL REPRESENTATION IN DESCRIPTION LOGICS

Our proposed formal framework borrows from Semantic Web languages and particularly Description Logics (Baader et al., 2003). 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, 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 conjunction $C \cap D$ 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 elements explicit. Concepts can be used in inclusion assertions $C \sqsubseteq D$, 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 containment: an interpretation $I$ satisfies an inclusion $C \sqsubseteq D$ if $C^I \subseteq D^I$, and satisfies a definition $C \equiv D$ when $C^I \equiv D^I$. A DL theory, also known as TBox or ontology, is a set of inclusion assertions and definitions. A model of a TBox $\mathcal{I}$ is an interpretation satisfying all inclusions and definitions of $\mathcal{I}$. 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 $\mathcal{ALCH}$ (the logic $\mathcal{AL}$ plus complex concept negation and role hierarchy).

Given the syntax and semantics of $\mathcal{ALCH}$, we are able to model the structure and integrity constraints of feature models in the $\mathcal{ALCH}$ Description Logic, hence it is expressive enough for our purpose. In the following similar to (Wang, Li, Sun, Zhang and Pan, 2007), 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 $SKB$, consists of the description of the hierarchical 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 Feature as the parent of all available features, and also let hasFeature be the core role that relates parent features with their child features. (It is important to note that in the following formalization Feature and hasFeature are actual constructs in the formalism metamodel whereas Child, Child₁, and Parent are placeholders for feature names that would be replaced at the metamodel instantiation phase. This can be seen in actual examples later).

\[
\text{Feature} \sqsubseteq \text{T} \\
\text{hasFeature} \sqsubseteq \text{ObjectProperty}
\]

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

  \[
  \text{Child} \sqsubseteq \text{Feature} \\
  \text{Parent} \sqsubseteq \text{Feature} \\
  \text{Parent} \sqsubseteq \exists \text{hasFeature}.\text{Child}
  \]

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

  \[
  \text{Child}_i \sqsubseteq \text{Feature}, \text{for } 1 \leq i \leq n \\
  \text{Parent} \sqsubseteq \text{Feature} \\
  \text{Parent} \sqsubseteq (\exists \text{hasFeature}.\text{Child}_1) \land \ldots \land (\exists \text{hasFeature}.\text{Child}_n)
  \]

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

  \[
  \text{Child}_i \sqsubseteq \text{Feature}, \text{for } 1 \leq i \leq n \\
  \text{Parent} \sqsubseteq \text{Feature} \\
  \text{Parent} \sqsubseteq (\exists \text{hasFeature}.\text{Child}_1) \land \ldots \land (\exists \text{hasFeature}.\text{Child}_n)
  \]

- **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:

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

\[114\] Journal of Research and Practice in Information Technology, Vol. 43, No. 2, May 2011
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 1 can be represented using an or feature group:

\[ \text{Shipping} \subseteq \text{Feature}, \quad \text{Pickup} \subseteq \text{Feature}, \quad \text{E} \rightarrow \text{delivery} \subseteq \text{Feature} \]
\[ \text{Airmail} \subseteq \text{Feature}, \quad \text{RegularMail} \subseteq \text{Feature} \]
\[ \text{Shipping} \subseteq (\exists \text{hasFeature}. \text{Pickup}) \cup \ldots \cup (\exists \text{hasFeature}. \text{RegularMail}) \]

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

\[ \text{FraudDetection} \subseteq \text{Feature}, \quad \text{Manual} \subseteq \text{Feature}, \quad \text{Automatic} \subseteq \text{Feature}, \quad \text{Manual} \subseteq \neg \text{Automatic} \]
\[ \text{FraudDetection} \subseteq (\exists \text{hasFeature}. \text{Manual}) \cup (\exists \text{hasFeature}. \text{Automatic}) \]

Finally, for the feature model root in addition to its feature relationships, we define the root as:

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

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

\[ \text{E} \rightarrow \text{Store} \subseteq (\exists \text{hasFeature}. \text{Feature}) \]
\[ \text{E} \rightarrow \text{Store} \subseteq (\exists \text{hasFeature}. \text{Tax}) \cap (\exists \text{hasFeature}. \text{Password}) \cap (\exists \text{hasFeature}. \text{Payment}) \]

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

### 3.2. Integrity Constraints

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

- **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:

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

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

\[ \exists \text{hasFeature}. \text{Cash} \subseteq (\exists \text{hasFeature}. \text{Pickup}) \]

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

\[ \exists \text{hasFeature}. \text{BaseFeature} \subseteq \neg (\exists \text{hasFeature}. \text{ExcludedFeature}) \]

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

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

All integrity constraints will be stored and represented through the integrity constraint base ($\mathcal{IC}$).
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 airmail 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 $\mathcal{SR}_h$.

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

- $\mathcal{SKB}$: The feature model structural knowledge, i.e. the definition of features and the feature model hierarchical tree structure, is represented using DL concepts and role relations modeled as described in Section 3.1.
- $\mathcal{JC}$: Integrity constraints, i.e. the includes and excludes relations between features, are modeled using subsumption assertions as described in Section 3.2.
- $\mathcal{SR}_h$: Stakeholder’s hard constraints, i.e. features that are requested by the stakeholders to be present in final configuration, are represented using 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 it 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 2 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 3. 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 3, 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. The development and representation of feature models in a modular manner provides many benefits and advantages, such as:

- **Distributed/collaborative development:** Given that the modular formalism for feature models allows different aspects of a feature model to be encapsulated into several different feature modules, it is possible for the design team to divide the domain of discourse into separate subdomains. Once the subdomains are identified, different teams can work on each of these subdomains as a result of which several feature modules are developed, each of which constitutes the features of one of the subdomains. The various design teams can then collaboratively work together to integrate these feature modules into a comprehensive modular feature model. The other possibility that is offered is that since the teams are working on their own subdomain and feature module, they can work independently of each other in a distributed setting.
Feature model maintenance and evolution: One of the important issues that designers face is the need for continuous update and refinement of their models referred to as model maintenance and evolution. Given the fact that within the modular feature model formalism, feature modules focus on the features of a specific subdomain of discourse and hence are independent of each other from the perspective of the subdomain that they cover, designers can update or maintain each feature module independently of the other modules. Having said that, this does not mean that changes within a feature module do not affect the modular feature models that contain this feature module, but rather it means that changes in one feature module will be contained in that module and will not propagate to other modules and hence it enhances the process of evolution and maintenance;

Module reusability: Generally, product line engineering is one of the important approaches for re-use base engineering. Modular feature models further enhance reusability by providing feature modules. As mentioned earlier, feature modules capture the features of a specific subdomain of discourse and can be used in the creation of any larger modular feature model that covers that specific subdomain. For this reason, each feature module does not necessarily need to be used only in one modular feature model and can be reused in different contexts for other purposes as well. Therefore, any one feature module can be reused in the construction of many other modular feature models. Besides reusability, it can indirectly increase productivity, promote knowledge transfer between projects, and reduce the likelihood of repeating design mistakes.
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 be integrated such that they form a large modular feature model.

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

In order to unambiguously refer to the features in different feature modules, they are prefixed with the index of the feature module, e.g., \( \text{pickup \_i} \) 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 module bridges.

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

\[
\begin{align*}
&i : X \leftrightarrow j : Y, \text{ an into — module bridge;} \\
&i : X \leftrightarrow j : Y, \text{ an onto — module bridge.}
\end{align*}
\]

where \( X \) and \( Y \) are concepts or roles from \( \mathcal{T}_i \) and \( \mathcal{T}_j \), 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 \( i : X \leftrightarrow j : Y \) 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 3. 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: \( \exists \text{hasFeature.Cash} \xleftrightarrow{\mathcal{E}} \exists \text{hasFeature.Pickup} \)

3: \( \exists \text{hasFeature.E delivery} \xrightarrow{\mathcal{C}} 1 : \exists \text{hasFeature.CreditCard} \)

1: \( \exists \text{hasFeature.CreditCard} \xrightarrow{\mathcal{C}} 4 : \exists \text{hasFeature.SpecialCharPassword} \)

2: \( \neg (\exists \text{hasFeature.UniversalTaxRate}) \xrightarrow{\mathcal{D}} 3 : \exists \text{hasFeature.Pickup} \)

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.
Modular Feature Models: Representation and Configuration

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:

1. $\text{GenericFeature} \subseteq T$
2. $\text{hasGenericFeature} \subseteq \text{ObjectProperty}$
3. $\text{GenericProductLine} \subseteq \exists \text{hasGenericFeature}.\text{GenericFeature}$
4. $\text{OurProductLine} \subseteq \text{GenericProductLine}$

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. 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. This activity, depending on the software product line engineering methodology, can have a centralized or distributed control. Different methodologies for distributed development of software product lines have been analyzed by Bosch and Bosch-Sijtsema (2010). Later, we will also show that stakeholder requests, $\mathcal{R}_k$, 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 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 $\langle F, B, S \rangle$ where $F = \{F_i\}_{i \in I}$ represents the individual feature modules, $B = \{B_{ij}\}_{i \in I}$ denotes the set of all available module bridges, and $S$ is the feature model sentinel.

Example 2. Let us consider the feature modules depicted in Figure 3. A modular feature model developed from these four individual feature modules consists of a feature model sentinel $S$, four feature modules $F_i$, $i = 1..4$, and several feature bridges including those described in Example 1.

As can be seen in Figure 4, the sentinel sits at the centre 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 $\mathcal{B}_S$. Each of the modules between the sentinel and the other feature modules ($\mathcal{B}_S$) consists of the following axioms (not shown in the figure, in other words the arrow labeled $\mathcal{B}_S$ will be carrying the following axioms):

$$\text{i: Feature} \rightarrow S : \text{GenericFeature}$$

$$\text{i: hasFeature} \rightarrow S : \text{hasGenericFeature}$$

Also the sentinel is given access to the top feature(s) of each module through a module bridge ($\mathcal{B}_S$) in order for us to be able to build the overall structure of the modular feature model in the sentinel. For instance, $\mathcal{B}_S$ consists 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 module bridges, a complete feature model can be put together in the sentinel for e-store as follows:

$$E \leftarrow \text{store} \equiv \exists \text{hasGenericFeature.Payment} \land \exists \text{hasGenericFeature.Tax} \land \exists \text{hasGenericFeature.Password} \land \exists \text{Shipping \equiv GenericFeature}$$

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.

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.

### 4.2. 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** \( T \models D \subseteq C \). 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 Description Logic TBox \( T \), which conveys
a software product line description, such as that of the electronic online store, \( PL = \mathcal{SKB} \cup \mathcal{HC} \), 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 using
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 through the
satisfiability of concept conjunction; 3) the feature model configuration is not at all compatible with
the feature model by 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 5.** Given a set of module bridges \( \mathcal{B}_{ij} \) from feature module \( F_i \) to \( F_j \), an operator \( \mathcal{B}_{ij}(\cdot) \),
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 (Serafini and Tamilin, 2009). 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.

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 (2007) for distributed Web service composition. 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 is 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 configuration algorithm for modular feature models (in this section, ‘feature models’ and ‘modular feature models’ are used interchangeably), we base our work on the notion of concept covering as presented in Ragone et al (2007). It is important to note that existing feature model configuration algorithms all only work with monolithic feature models, whereas the following proposed algorithm is designed to work with modular feature models. The following sections introduce 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 model configuration, denoted $C$, is consistent with the structural knowledge of a feature model, 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 $T \not\models PL \cap C \subseteq \bot$ 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 Di Noia, Di Sciascio, Donini and Mongiello (2003), to provide an explanation hypothesis when subsumption does not hold.

Definition 6. 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 $L, S, D, T$, 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 di Noia et al (2003), the minimality criteria for $H$ and a polynomial algorithm, to find solutions which are irreducible, for $ALN$ DL, have been proposed. More recent work has extended support for CAP in $SH$ DL (di Noia, Sciascio and Donini, 2009), which is an extension of $ALCH$ DL that we use in this paper ($ALCH \subset SH$) and can hence be applied to our specific problem.

As described in Colucci, di Noia, Sciascio, Donini and Mongiello (2005), 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 $S$?
Modular Feature Models: Representation and Configuration

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 $\mathcal{SR}_b$, and some feature model configuration $\mathcal{C}$ 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 $\mathcal{C}$ needed to completely satisfy a stakeholders’ requests in $\mathcal{SR}_b$. 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 $\mathcal{C}$ with respect to $\mathcal{SR}_b$.

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 to compute concept covering, or in other words feature model configurations based on stakeholders’ requests, in terms of concept abduction (Colucci, Di Noia, Di Sciascio, Donini and Ragone, 2005; Colucci, di Noia, Di Sciascio, Donini and Ragone, 2005). 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 7.** (Ragone et al., 2007) Let $\mathcal{D}$ be a concept, $\mathcal{R} = \{S_1, S_2, \ldots, S_k\}$ be a set of concepts, and $\mathcal{T}$ be a set of axioms, all in some DL $\mathcal{L}$, e.g., $\mathcal{L} = ALC^H$, where $\mathcal{D}$ and $S_1, \ldots, S_k$ are satisfiable in $\mathcal{T}$. Let also $\prec_T$ be an order relation over the concepts in $\mathcal{L}$ taking into account the TBox $\mathcal{T}$. The Concept Covering Problem (CCoP) denoted as $\mathcal{V} = \langle L, R, D, T \rangle$ is finding a pair $\langle \mathcal{R}_c, H \rangle$ such that

1. $\mathcal{R}_c \subseteq \mathcal{R}$, and $\mathcal{T} \not\prec_T \bigwedge_{S_i \in \mathcal{R}_c} S_i \equiv \mathcal{L}$;
2. $H$ is a solution to $\langle L, S, D, T \rangle$, and $H \prec_T \mathcal{D}$.

We call $\langle \mathcal{R}_c, H \rangle$ a solution for $\mathcal{V}$, and say that $\mathcal{R}_c$ covers $\mathcal{D}$. Finally, we denote $SOLCCoP(\mathcal{V})$ the set of all solutions to a CCoP $\mathcal{V}$.

In the above definition, the elements for the solution $\langle \mathcal{R}_c, H \rangle$ of a CCoP represent respectively:

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

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

Di Cugno, Noia, Sciascio and Ragone (2006) 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 (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 $\mathcal{R}_c$, 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 $\mathcal{R}_c$. This fact gives way to the definition of best cover and exact cover.

**Definition 8.** (di Noia, Sciascio and Donini, 2004) A best cover for a CCoP $\mathcal{V}$, w.r.t. an order $\prec_T$ on the adopted DL, is a solution $\langle \mathcal{R}_c', H' \rangle$ for $\mathcal{V}$ such that there is no other solution $\langle \mathcal{R}_c'', H'' \rangle$ for $\mathcal{V}$ with $H' \prec_T H_b$.

There is no solution \(< R', H' >\) for \(V\) such that \(H'\) is ‘smaller’ than \(H_b\) with respect to an order \(\prec_{\tau}\). Observe that, since the solution for concept abduction is not unique in general, there could be two solutions \(< R', H_1 >, < R', H_2 >\) 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 \(\prec_{\tau}\) on the adopted DL. However, for our purpose the SIM-DL algorithm presented in Janowicz (2007) can be used to compute an order \(\prec_{\tau}\) for Description Logic \(\mathcal{ALCN'R}\). SIM-DL can be used for our purpose since we use \(\mathcal{ALCH}\) which is a subset of \(\mathcal{ALCN'R}\).

**Definition 9.** (di Noia et al., 2004) An exact cover for a CCoP \(V\) is a solution \(< R_c, H_e >\) for \(V\) such that \(T \models H_e \equiv T\).

### 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 10.** (Extends Definition 7) Let \(S\mathcal{R}_h\) be the stakeholders’ requests, \(F = S\mathcal{K}B \cup S\mathcal{C}\) be a given modular feature model defined in terms of its structural knowledge and integrity constraints, and \(T\) be a set of axioms used to define \(F\), all in a DL \(L\). Let also \(\prec_{\tau}\) be an order relation over \(L\) taking into account the TBox of \(F\). The modular feature model configuration problem is defined as CCoP \(V = < L, F, S\mathcal{R}_h, T >\). All possible feature model configurations for modular feature model \(F\) based on \(S\mathcal{R}_h\) are attained by SOLCCoP(\(V\)).

Following from this correspondence, a solution \(< C, H >\) for \(V = < L, F, S\mathcal{R}_h, T >\) 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 \(S\mathcal{R}_h\) 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 \(< C, H >, H \equiv T\). This means that all members of \(S\mathcal{R}_h\) are present in \(C\). However, if such a configuration does not exist, a configuration \(< C, H >\), where for any other possible derivable configurations such as \(< C', H' >\), we have \(H \prec_{\tau} H'\), 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 Ragone et al. (2007) making it suitable for feature model configuration, building on and extending a classical greedy set covering one (Cormen, Leiserson and Rivest, 1990). The algorithm takes as input the structural knowledge and integrity constraints.
Modular Feature Models: Representation and Configuration

of a modular feature model along with a set of specific stakeholder requests, and develops a corresponding configuration.

Algorithm BuildConfiguration($F, S_R_h, T$)

input concepts $S_R_h, S_i \in F$ and $F = U_{i \in S} (\cdot) \cup S$, where $S_R_h$ and $S_i$ are satisfiable in $T$ and $S$ is the feature model sentinel

output $< C, H >$

begin algorithm

/* [\triangle] Add all of the implications of the stakeholder requests to $S_R_h$ */

foreach ($f \in \exists hasFeature.f \in F$

if $f \in S_R_h$ then

$S_R_h = S_R_h \land f$

end if

done foreach

$C = \emptyset$

$S_R_{hu} = S_R_h$

$H_{min} = S_R_h$

Do

$S_{MAX} = T$

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

foreach $S_i \in F$

if ($C \cup \{S_i\}$ is a cover for $S_R_{hu}$ and $(T \not= C \cup \{S_i\} \not= I)$ then

//SolveCAP solves a concept abduction problem.

$H = solveCAP(< L, S_i, S_R_{hu}, T >)$

/* [\spadesuit] Choose $S_i$ based on an order $<_T$ */

if $H <_T H_{min}$ then

$S_{MAX} = S_i$

$H_{min} = H$

end if

done foreach

/* [\heartsuit] If a new $S_i$ is found then add $S_i$ to C and remove it from $F$ */

if $T = S_{MAX} \not= T$ then

$F = F \setminus \{S_i\}$

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

$S_R_{hu} = H_{min}$

/* [\star] Enforce modular feature model integrity constraints */

foreach ($\exists hasFeature.f \in \exists hasFeature.f \in F$

if $f \in C$ then

$F = F \setminus \{f\}$

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

end if

end
This algorithm attempts to create a configuration that covers $\mathcal{SR}_h$ as much as possible, 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 $[\Delta]$. Then, $BuildConfiguration$ adopts a greedy approach towards the selection of the most appropriate features to be included in the developed configuration $[\bullet]$, 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 $\mathcal{SR}_h$ $[\triangledown]$. Once such a feature is identified, it is added to the configuration and removed from the list of stakeholder constraints $[\bigcirc]$. 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 $[\bigtriangledown]$. This process is repeated until no more appropriate features can be added to the configuration. Finally, $BuildConfiguration$ 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 an exact cover was not achieved is presented in $\mathcal{SR}_hu$.

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 1 has been modularized and represented as depicted in Figures 3 and 4. 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.

6.1. Case Study 1

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:

$\mathcal{SR}_h \equiv e \setminus store \cap Credit\cap Pickup \cap Never$

In order to form a configuration of the modular electronic store feature model, we need to employ the $BuildConfiguration$ algorithm. The algorithm evaluates the stakeholders’ requests in
Modular Feature Models: Representation and Configuration

several rounds and creates a desirable configuration given the requests. So given $\mathcal{SR}_h$, we show in the following just the first round of the 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:

- (Δ) To formulate the configuration, BuildConfiguration first extends $\mathcal{SR}_h$ with all of the child features of the requested features, which were originally in $\mathcal{SR}_h$. As it can be seen, it is only e-store that has child features in this example, so the expansion of e-store in $\mathcal{SR}_h$ entails:

$$\mathcal{SR}_h \equiv e \text{ – store} \cap \text{CreditCard} \cap \text{Pickup} \cap \text{Never} \cap \text{Payment} \cap \text{Tax} \cap \text{Password}$$

- Now, the new features in the expanded $\mathcal{SR}_h$ need to be expanded; hence, Password and Tax should be unfolded. This process is performed until no further expansion is possible.

- (ϕ) The second step consists of a greedy search between the modular feature model features to find the one that provides the most covering over $\mathcal{SR}_h$.

- (ρ) 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 $\prec_f$. It is easy to see that since the selection of the CreditCard 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 CreditCard, can be added to the configuration that is being gradually developed. Therefore, $\mathcal{C} = \{\text{CreditCard}\}$. 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 AutomaticFraudDetection and SpecialCharacterPasswords 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 $\mathcal{C}$ are enforced, so now because the AutomaticFraudDetection and SpecialCharacterPasswords features are in $\mathcal{C}$ their integrity constraints will also be enforced if any.

- (φ) 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.

6.2. Case Study 2
Let us now analyze the execution time performance of the modular feature model configuration process in contrast with monolithic feature models.

---

Journal of Research and Practice in Information Technology, Vol. 43, No. 2, May 2011
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 (Mendonca, Branco and Cowan, 2009), which was recently used in the research community for evaluating feature model configuration methods and the execution time evaluations are based on the framework provided by Boskovic, Bagheri, Gasevic, Mohabbati, Kaviani and Hatala (2010). The main reason why we chose synthetic feature models for our work is that many of the real world feature models that are designed within the industry are proprietary and are not shared with the academic community. The only shared feature models that are available through the SPLOT repository are rather small feature models that are not really suitable for showing the performance of the proposed modular feature model configuration algorithm. Hence, we chose the SPLOT Feature Model Generator to create large feature models based on the characteristics of real world feature models that would allow us to analyze the performance of our proposals.

The scenarios for our four experiments that we have performed are described as follows:

**Scenario 1** –

**Purpose:** The goal of the first scenario was to analyze if modularization enhances the execution time of the configuration process. In other words, would the time needed for configuring a modular feature model be less than that needed for configuring the same feature model but with a monolithic design?

**Procedure:** For this scenario, several feature models with exactly 2000 features were generated using the SPLOT feature model generator tool and were each partitioned into a different number of modules. Modularization was performed such that features with more integrity constraints were kept together as much as possible. The number of requests from the stakeholders was set to 20. The generated feature models were configured based on the algorithm proposed in this paper and the average configuration running time was calculated.

**Observation:** Figure 5a shows that the configuration process for a feature model with only one module (monolithic feature model) takes much longer in terms of running time compared to the other cases (i.e., modular feature models with 2000 features and a varying number of modules with more than one module).

**Interpretation:** This observation shows that given the same number of features in a feature model, 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 in Figure 5a. This can be due to the fact that 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. The feature model designers would be the best for deciding on the best number of modules for the modular feature model.

Scenario 2 –

**Purpose:** In the second scenario, the goal was to analyze whether an increase in the number of features in a given feature model effects the configuration execution time or not. In other words, does the number of features in a modular feature model impact the configuration time?

**Procedure:** For this purpose, different feature models were generated using the SPLOT feature model generator, each which of consisted of a different number of features. Each of these feature models were partitioned into numerous modules. Similar to the first scenario, the number of stakeholder requests was set to 20. These feature models were then configured and the average running time was recorded.

**Observation:** Figure 5b depicts the execution time of the configuration algorithm on these different feature models. It can be seen that as the number of features increases the configuration execution time also increases.

**Interpretation:** The important observation from 5b is that although the configuration execution time rises with the increase in the number of features, but at the same time a comparison between the modular feature models with the same number of features shows that those modular feature models that have a higher number of modules have a smaller configuration execution time.

Scenario 3 –

**Purpose:** The intention of the third scenario is to investigate whether the increase in the number of stakeholder requests has any substantial effects on the configuration execution time of similar feature models or not. In other words, does the increase in the number of requests from the stakeholders for the same feature model impact the configuration execution time?

**Procedure:** In the previous scenarios, the number of stakeholder requests was set to a constant of 20. However, in this third scenario, the number of features was set to a constant of 2000 in all of the feature models, but the number of requests changed in the different experiments. Therefore, different feature models all with 2000 features were generated and modularized. Subsequently, the configuration algorithm was run on these modular feature models with varying number of stakeholder requests. The average running time was recorded.

**Observation:** As it can be seen in Figure 6a, 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.

**Interpretation:** From the observation, we can conclude that modularity of feature models keeps the configuration time robust to changes in the number of requests from the stakeholders. This can be due to the fact that the requests will be divided between the different modules for satisfaction in the
Modular Feature Models: Representation and Configuration

Figure 6: Evaluating feature model execution time: a) Scenario 3 on left; b) Scenario 4 on right.

It is interesting to see that although the number of features and feature modules has gradually grown, the configuration time has stayed relatively the same in the experiments; however, in the monolithic feature model, with the increase in the number of features, the execution time grows much higher.

Interpretation: The stability in the execution time of the configuration algorithm on modular feature models with varying numbers of features and feature modules and the increase in the execution time of monolithic feature models is an indication of the fact 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.

Scenario 4 –
Purpose: In Scenario 4, the goal was to analyze the execution time performance of the configuration algorithm given varying numbers of features and feature modules. In other words, would the running time of the algorithm be impacted as the number of features and feature modules increase in the modular feature model or not.

Procedure: As shown in Figure 6b, we increased both the number of modules and the number of available features in different experiments. The number of stakeholder requests was kept at a constant of 20. Different feature models with varying numbers of features were generated and modularized. The configuration algorithm was run on these feature models and the average time was calculated.

Observation: It is interesting to see that although the number of features and feature modules has gradually grown, the configuration time has stayed relatively the same in the experiments; however, in the monolithic feature model, with the increase in the number of features, the execution time grows much higher.

Interpretation: The stability in the execution time of the configuration algorithm on modular feature models with varying numbers of features and feature modules and the increase in the execution time of monolithic feature models is an indication of the fact 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.

It is important to analyze why modularization impacts the configuration time and manages to tame it under different scenarios. From our point of view, the main reason behind this is because in the proposed configuration algorithm that has been introduced in this paper only those modules of the feature model are included in the configuration process, whose features are required to be present in the final feature model configuration. For instance, if a feature model has 10 modules, it is possible that the features of only 5 of its modules will be included in the configuration process. Therefore, since this process removes some of the extra modules that are not relevant to the configuration process at hand, therefore, the configuration execution time will be much lower. On the other hand, within monolithic feature models since all of the features will be included in any
configuration process and even the superfluous ones will not be ignored, the average running time of the configuration algorithm is higher than their modularized counterparts.

The other parameter that could play an important role in the configuration time of a modular feature model is the distribution of stakeholder requests among the available modules. For instance, if the entire stakeholder requests concern only one module, then the configuration time will be much smaller compared to the case when the stakeholder requests cover all of the available modules. It is worth mentioning that when the stakeholder requests include features from all of the feature model modules, then the configuration execution time of a modular feature model is equivalent to the configuration execution time of the equivalent monolithic feature model.

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. Furthermore, collaborative development process facilitates discovery and repair in divergence and inconsistency in terminology can be discovered and resolved. 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 (Wang et al., 2007) has shown promising prospects 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 $\varepsilon$-connections (Kutz, Lutz, Wolter and Zakharyaschev, 2004), Interface-based Formalism (IBF) (Ensan and Du, 2008), Distributed Description Logic (DDL) (Borgida and Serafini, 2003), and Package-based Description Logics (P-DL) (Bao, Caragea and Honavar, 2006). Each of these formalisms provides 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 (Ensan and Du, 2008) the attention is more on knowledge encapsulation and inheritance, which are not the main concerns in modular feature models; 2) $\varepsilon$-connections (Kutz et al., 2004) 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 (Bao et al., 2006) 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 tool suite, called Drago (Drago, 2011). Drago is a peer-to-peer reasoning platform, built on top of Pellet OWL Reasoner (Pellet, 2011) that supports visualization and standard reasoning mechanisms (subsumption and concept satisfiability) over distributed and connected ontologies represented in OWL. Therefore, 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 7 depicts the visualization of the connection of a feature module with the sentinel in Example 2. Modular feature models are represented as OWL ontologies, and module bridges are shown as lines.
connecting the OWL representations of feature modules, which make 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 (Fenton and Pfleeger, 1997), 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 Cormen et al (1990) 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 BuildConfiguration algorithm, the complexity source is in the solution of the CAPs and the ordering performed in \[ \mathcal{O} \]. It is possible to show that 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
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 Build Configuration. The algorithm returns a good configuration, which may not always be the best one.

It is worth noting that although the underlying formalism for representing modular feature models has been Description Logics in this paper, in reality, feature model modules can be simply developed using any feature modeling tool such as SPLIT (Mendonca et al., 2009), and then we automatically convert these feature modules into Description Logic format (OWL ontologies) without the need for the designers to be aware of this or be familiar with Description Logics in any way. Software product line designers will only interact with feature model design tools and the conversion to and from Description Logics is automatically handled behind the scenes. For our research, we have used SPLIT and Drago for implementation and testing purposes in terms of creating, manipulating, configuring and visualizing modular feature models introduced in this paper.

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 (Moskewicz, Madigan, Zhao, Zhang and Malik, 2001). 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 (2002) was the first to propose the adoption of propositional formula for formally representing software product lines. This idea has been extended by creating a connection between feature models, grammars and propositional formula by Batory (2005). Grammars have also been used in other work such as in the work of Czarnecki et al (2004) 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 (2005) employ the Object Constraint Language (OCL) to express such additional restrictions. In their approach, feature models are converted into UML diagrams where integrity constraints are enforced on them through OCL expressions.

Wang et al (2007) 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. 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++ (Tsarkov and Horrocks, 2006) 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 initial 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, Kleinermann ans De Troyer (2009) employs description logic ontologies and Semantic Web rules expressed in SWRL (Horrocks, Patel-Schneider, Boley, Tabet, Grosof and Dean, 2004) 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 (Schobbens et al., 2006; Janota and Kiniry, 2007; Benavides, Segura, Trinidad and Ruiz-Cortés, 2007). Three of the most widely used methods in this area are Constraint Satisfaction Problem (CSP) solvers (Benavides, Trinidad and Ruiz-Cortés, 2005), propositional SATisfiability problem (SAT) solvers (Batory, 2005; Bagheri, Noia, Gasevic and Ragone, 2011), and the Binary Decision Diagrams (BDD) (Mendonca, Wasowski, Czarnecki and Cowan, 2008). 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 (Benavides, Segura, Trinidad and Ruiz-Cortés, 2006). 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 (Benavides et al., 2007). Closely related is the employment of Alloy (Jackson, 2002) for analyzing the properties of feature models that are based on satisfiability of first-order logic specifications converted into Boolean expressions (Gheyi, Massoni and Borba, 2006). 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 compensate for their space complexity.

Furthermore, Czarnecki, She and Wasowski (2008) have developed probabilistic feature models where a set of joint probability distributions over the features provide the possibility for defining hard and soft constraints. The joint probability distributions are mined from existing software 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 $P(N)$, which is standard Propositional Logic extended with concrete domains (Bagheri et al., 2011) 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. $P(N)$ 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 Logics 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 much positive implication 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.

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

1. In this paper, we have only provided the formalism for representing modular feature models along with an appropriate algorithm that would take a modular feature model and a set of stakeholder requests as input and would generate a feature model configuration as output. One of the main directions that we intend to study in the future work is how can feature models be actually modularized. This is an important issue to study as it pertains to many important matters such as the suitable number of modules in a modular feature model, and the proper module for the placement of each feature. We will be exploring two options to address this either 1) by the design of a support methodology for developing modular feature models, or 2) through the development of an algorithm for automatically modularizing a monolithic feature model;
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 (Zhang, Deng, Wu, Zhou and Ma, 2008). These metrics are essentially analogous to those devised as code or object-oriented design metrics (Basili, Briand and Melo, 1996). 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;

3. The other aspect that we intend to carefully investigate in our future work is to empirically compare our modular feature model configuration algorithm, which is based on Description Logics with other existing feature model configuration algorithms that are based on Propositional Logic and SAT solvers. Currently the major advantage of our work over those existing methods is that it fully supports for configuration of modular feature models, whereas the other methods only work with monolithic feature models. However, we will need to also compare their performance in terms of running time.

ACKNOWLEDGEMENT

The research results present in this paper are supported in part by the Alberta Innovates Technology Futures through the New Faculty Award program.

REFERENCES


MODULAR FEATURE MODELS: REPRESENTATION AND CONFIGURATION


Modular Feature Models: Representation and Configuration


BIOGRAPHICAL NOTES

Ebrahim Bagheri (http://ebagheri.athabascau.ca/) is currently an assistant professor at the AU School of Computing and Information Systems, a visiting professor with the University of British Columbia, honorary research associate at the University of New Brunswick and an IBM CAS faculty fellow. Previously, he was affiliated with the Institute for Information Technology at the National Research Council Canada. Ebrahim completed his PhD degree at the Faculty of Computer Science, University of New Brunswick, and specializes in topics ranging from the meta-modeling of complex interconnected systems to collaborative information systems design. Currently, his research focuses on two areas namely, quality engineering for software product lines; and social informatics for healthcare. His work on collaborative modeling is one of a kind in providing tools and techniques for collaborative risk management and quality engineering. He has extensively published over 80 papers in top-tier international journals and conferences including the IEEE Transaction on Systems, Man, and Cybernetics Part A, Elsevier journal of Systems and Software, Springer’s Software Quality Journal, Wiley’s Journal of Software Maintenance and Evolution: Research and Practice, Springer’s Information Systems Frontiers Journal, Springer’s Knowledge and Information Systems Journal among others.
Modular Feature Models: Representation and Configuration

Faezeh Ensan joined the Sauder School of Business at the University of British Columbia (UBC), as a Postdoctoral Fellow in 2011. Before joining UBC, she spent four years completing her PhD in computer science at the Faculty of Computer Science, University of New Brunswick, Canada. She received her Masters of Engineering from the Ferdowsi University of Mashhad, and her Bachelors of Computer Engineering from the Faculty of Engineering, University of Tehran in 2004. Her research is focused on aspects of knowledge engineering, conceptual modeling, ontology development, modular ontologies and belief revision.

Dragan Gašević (http://dgasevic.athabascau.ca) is a Canada Research Chair in Semantic Technologies and an associate professor in the School of Computing and Information Systems at Athabasca University. He is also an Adjunct Professor in the School of Interactive Arts and Technology at Simon Fraser University, an IBM CAS faculty fellow, and an associated research member of the GOOD OLD AI Research Network at the University of Belgrade. His research interests include semantic technologies, software language engineering, technology-enhanced learning, and service-oriented architectures. Being a principal investigator in several national and international research projects, he is a recipient of Alberta Innovates Technology Futures’ 2008 New Faculty Award for his research on families of semantically-enhanced business processes and service-oriented systems. A (co-)author of numerous research papers and often keynote speaker, he is the lead author of the book monograph Model Driven Engineering and Ontology Development published in two editions by Springer in 2006 and 2009. Committed to the development of international research community and the concept of cross-community engineering, Dragan has (co-)founded several new events (e.g., SLE and LAK), has served as a chair and member of numerous steering and program committees, and edited several journal special issues and books.

Marko Bošković (http://mboskovic.athabascau.ca/) is a graduate from the Military Academy, Belgrade, and holds a PhD from the University of Oldenburg, Germany. Currently he is a postdoc fellow in the School of Computing and Information Systems at Athabasca University, and School of Interactive Arts and Technology at Simon Fraser University, Canada. He is researching in the area of integrating Semantic Web Technologies in the process of development of Software Product Lines, as well as integration of non-functional properties assessment techniques in the development of Service Oriented Software Product Lines. His research interests are integration of non-functional system requirements estimation and evaluation in Domain Specific Modeling Languages, Software Product Lines and Model-driven Engineering in general. He served as reviewer for several journals and international conferences and workshops. He is on the editorial board of one electronic journal.