A Semantic Web Approach to Feature Modeling and Verification

Hai Wang[1] Yuan Fang Li[2]
Jing Sun[3] Hongyu Zhang[4] Jeff Pan[1]

[1] The University of Manchester, United Kingdom
{hwang,pan}@cs.man.ac.uk
[2] National University of Singapore, Singapore
liyf@comp.nus.edu.sg
[3] The University of Auckland, New Zealand
j.sun@cs.auckland.ac.nz
[4] RMIT University, Australia
hongyu@cs.rmit.edu.au

Abstract. Feature models are widely used in domain engineering to capture common and variant concepts among systems in a particular domain. However, the lack of a formal semantics of feature models has hindered the development of this area. This paper presents a Semantic Web environment for modeling and verifying feature diagrams using ontologies. We use OWL DL (a decidable dialect of OWL) to precisely capture the relationships among features in feature diagrams and configurations. OWL reasoning engines such as RACER are deployed to check for the inconsistencies of feature configurations fully automatically. As part of the environment, we also develop a CASE tool to facilitate the visual development, interchange and reasoning of feature diagrams represented as ontologies.

1 Introduction

Domain engineering is a software reuse approach that focuses on a particular application domain such as word processing, inventory management systems, etc. In domain engineering, we perform domain analysis and capture domain knowledge in the form of reusable software assets. By reusing the domain assets, an organization will be able to deliver a new product in the domain in a shorter time and at a lower cost. In industry, domain engineering forms a basis for software product line practices [14].

Feature modeling [3] plays an important role in domain engineering. Features are prominent and distinctive user visible characteristic of a system. Systems in a domain share common features and also differ in certain features. In feature modeling, we identify the common and variant features and capture them as a graphical feature diagram. Feature modeling is considered as "the greatest contribution of domain engineering to software engineering" [3].

Quite a number of feature-based reuse approaches have been proposed, such as FODA (Feature-Oriented Domain Analysis) [12], FORM (Feature-Oriented Reuse Method) [13] and FeatuRSEB [8]. However, due to the absence of a formal semantics of features and

feature modeling, there is no automated tool that can check the correctness of a particular feature configuration based on the constraints specified in a feature model.

Works in the description logics area also inspired us to apply Semantic Web technologies to feature modeling. For instance, various description logics systems have been applied to solving complex configuration problems [17, 6] and verifying the consistency of UML diagrams [1]. However, to our best knowledge, Semantic Web and ontology languages have not been applied to feature modeling problems before. Moreover, the Semantic Web approach that we adopt provides an semantic foundation and working environment that facilitates model creation, verification, integration, maintenance, etc.

Ontology languages such as OWL [11] play a key role in realizing the full potential of Semantic Web as they prescribe how data are defined and related. We feel that there is a strong similarity between Semantic Web ontology engineering and feature modeling, both of which represent concepts in a particular domain and define how various properties relate among them. Hence, we believe that Semantic Web can play important roles in domain engineering, and vice versa.

In this paper, we explore the synergy of domain engineering and Semantic Web. We propose methods for transforming a feature model into an OWL DL ontology. Being based on XML, the OWL format facilitates storage and exchange of the feature models. We then use OWL reasoning engine such as RACER [10] to perform automated analysis over an OWL representation of the feature model. The analysis helps us detect possible inconsistencies in feature configurations. We illustrate our approach using an example of the Graph Product Line (GPL) feature model, which is a standard problem proposed in [15] for evaluating product line technologies. Furthermore, we developed a CASE tool to facilitate visual development, reasoning and distribution of feature models in the Semantic Web environment.

The remainder of the paper is organized as follows. In Section 2, we give a brief overview of feature modeling and Semantic Web ontology languages and tools. Section 3 presents our proposal of feature model using OWL. We also show how Semantic Web reasoning engines, can be used to perform automated analysis over the OWL representation of the feature configurations. In Section 4, we demonstrate the CASE tool we developed to facilitate the visual creation and reasoning of feature models. Section 5 concludes the paper and describes future work.

2 Background Information

2.1 Feature Modeling - Preliminary

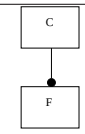
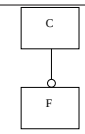
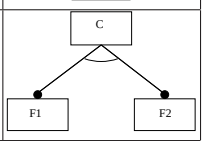
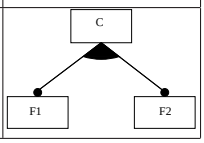
Concepts & Features

Feature research has its root in conceptual modeling and cognitive science. In classical conceptual modeling, we describe concepts by listing their features, which differentiate instances of a concept. In software engineering, software features differentiate software systems. Features of a software system are not only related to user-visible functional requirements, but also related to non-functional requirements (quality attributes), design decisions, and implementation details. In domain engineering and software product line context, features distinguish different members of a product line. A product line can

be seen as a concept, and members of the product line can be seen as instances of the concept. Product line members share common features and also differ in certain features.

Feature Diagrams & Feature Relations
Conceptual relationships among features can be expressed by a feature model as proposed in [12].

Table 1. Features types

Type	Notation
Mandatory	
Optional	
Alternative	
Or	

A feature model consists of a feature diagram and other associated information (such as rationale, constraints and dependency rules). A feature diagram provides a graphical tree-like notation that shows the hierarchical organization of features. The root of the tree represents a concept node. All other nodes represent different features.

Table 1 provides an overview of some commonly found feature types. The graphical notation introduced in [3] are used here. In Table 1, assuming the concept *C* is selected, we have the following definitions on its child features:

- Mandatory – The feature must be included into the description of a concept instance.
- Optional – The feature may or may not be included into the description of a concept instance.
- Alternative – Exactly one feature from a set of features can be included into the description of a concept instance.
- Or – One or more features from a set of features can be included into the description of a concept instance.

A feature diagram itself cannot capture all the inter-dependencies among features. We have identified two additional relations among features: *requires* and *excludes*.

- Requires – The presence of some feature in a configuration requires the presence of some other features.
- Excludes – The presence of some feature excludes the presence of some other features.

As the *Requires* and *Excludes* relations do not appear in a feature diagram, they are usually presented as additional constraints in a textual description.

The Graph Product Line (GPL) feature model

The Graph Product Line (GPL) example was proposed by Lopez-Herrejon and Batory as a standard problem for evaluating software product line technologies [15]. We use it as a case study to demonstrate the effectiveness of our approach in verifying feature models using OWL. The GPL is a family of classical graph applications in the Computer Science domain. Members of GPL implement one or more graph algorithms, over a directed or undirected graph that is weighted or unweighted, and one search algorithm if required [1]. We summarize it as follows.

GPL is a typical software product line in that different GPL applications are distinguished by a set of features. Lopez-Herrejon and Batory have identified the following features in GPL:

- Algorithms – A graph application implements one or more of the following algorithms: Vertex numbering (*Number*), Connected Components (*Connected*), Strongly Connected Components (*StronglyConnected*), Cycle Checking (*Cycle*), Minimum Spanning Trees (*MST*), and Single-Source Shortest Path (*Shortest*).
- Graph Type – A graph is either *Directed* or *Undirected*, and its edges can be either *Weighted* or *Unweighted*.
- Search – A graph application requires at most one search algorithms: Breadth-First Search (*BFS*) or Depth-First Search (*DFS*).

Based on the above feature classification, a feature diagram for the Graph Product Line (GPL) applications can be defined as shown in Figure 1.

Not all combinations of the features described in the above feature diagram (Figure 1) are valid in a GPL implementation. For example, if a graph application implements the Minimum Spanning Trees (MST) algorithm, we have to use the Weighted and Undirected graph types and require no search algorithm. Table 2 shows the additional constraints among the GPL features for representing a valid combination, adapted from Lopez-Herrejon and Batory [15].

From the GPL model presented in Fig. 1 above and additional constraints, we can see that *(GPL, GraphType, Directed, Unweighted, Algorithms, Number)* is a possible configuration derived from the *GPL* feature model. However, not all combinations of

[1] More information about the GPL example can be found online at: `http://www.cs.utexas.edu/users/dsb/GPL/graph.htm`

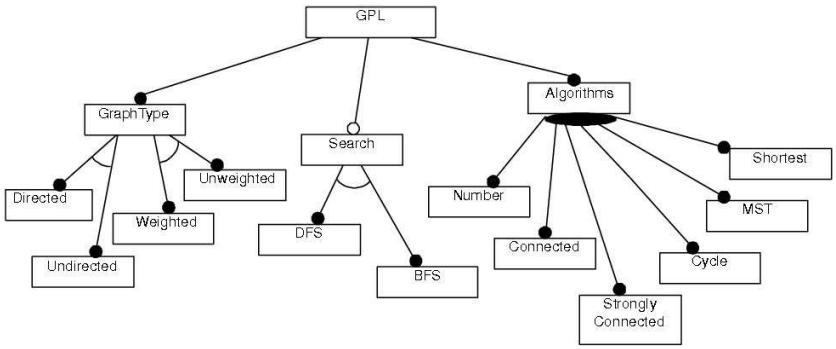


Fig. 1. A feature model for Graph Product Line.
Table 2. Additional Constraints on GPL *Algorithms*

Algorithm	**Searches Required**	**Required Graph Type**	**Required Weight**
Vertex Numbering	DFS, BFS	Directed, Undirected	Weighted, Unweighted
Connected Components	DFS, BFS	Undirected	Weighted, Unweighted
Strongly Connected	DFS	Directed	Weighted, Unweighted
Cycle Checking	DFS	Directed, Undirected	Weighted, Unweighted
Minimum Spanning Tree	None	Undirected	Weighted
Single-Source Shortest Path	None	Directed	Weighted

features are valid. For example, the configuration *(GPL, GraphType, Directed, Undirected, Weighted, Algorithms, Shortest)* is invalid since the features *Directed* and *Undirected* are exclusive to each other.

2.2 Semantic Web - Ontology languages & Tools

Description logics [19] are logical formalisms for representing information about classes, individuals and their relationships. It evolved from frame-based systems [18] and predicate logic and is well-known for the trade-off between expressivity and decidability.

Based on RDF Schema [4] and DAML+OIL [21], OWL [16] is the de-facto ontology language for the Semantic Web. It consists of three increasingly expressive sublanguages: OWL Lite, DL and Full. OWL DL is very expressive yet decidable. As a result, core inference problems, namely concept subsumption, consistency and instantiation, can be performed fully automatically. In OWL (and description logics), conceptual entities are organized as classes in hierarchies. Individual entities are grouped under classes and are called instances of the classes. Classes and individuals can be related by properties.

RACER, the **R**enamed **A**Box and **C**oncept **E**xpression **R**easoner [10], implements a TBox (terminological box, class-level) and ABox (assertional box, instance-level) reasoner for the description logic $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D})^-$ [9]. It can be regarded as:

- a Semantic Web inference engine,
- a description logic reasoning system capable of both TBox and ABox reasoning,
- a prover for modal logic Km.

In the Semantic Web domain, RACER's functionalities include developing ontologies (creating, maintaining and deleting concepts, roles and individuals); querying, retrieving and evaluating the knowledge base, etc. It supports RDF, DAML+OIL and OWL.

3 Feature Modeling using OWL

In this section, we use OWL DL to formally define the semantics of the above-mentioned feature relations, namely *mandatory*, *optional*, *alternative*, *or*; and additional constraints *requires* and *excludes*. With feature models expressed in OWL DL, a Semantic Web environment can be built to facilitate feature model storing, sharing and distribution and to assist design cooperation. In this paper, we only focus on verifying feature models using OWL.

Our presentation of the OWL encoding will be divided into three parts. In the first part, we present how a feature diagram and additional constraints are modeled in OWL. In the second part, the modeling of feature configurations are discussed. In the last part, we discuss the verification of feature configurations using the GPL example. The feature modeling in OWL is presented in a syntax similar to the "DL syntax" given in [11].

3.1 Conceptual Modeling

Pre-processing
Before we model the different feature relations in a feature diagram, we need to build the OWL ontology for the various nodes and edges in the diagram. The ontology is constructed in two steps. For a parent feature G and its child features $F_1, ..., F_n$, the initial modeling produces the following ontology.

1. We identify the nodes (concepts and features) in a feature diagram. Each node in the diagram is modeled as an OWL class (subclass of the top class $\top$). Moreover, we assert that these classes are mutually disjoint.

   $G \sqsubseteq \top$ $\qquad$ $G \sqcap F_i = \bot$, for $1 \leq i \leq n$
   $F_i \sqsubseteq \top$ $\qquad$ $F_i \sqcap F_j = \bot$, for $1 \leq i < j \leq n$

   Note that the clause $G \sqcap F_i = \bot$ is used to assert the disjointness of the classes G and F_i.
   Taking GPL as an example, concept GPL and features such as Search, DFS, Cycle, etc. are all classes in the ontology.

$GPL \sqsubseteq \top$	$GPL \sqcap GraphType = \bot$
$GraphType \sqsubseteq \top$	$GPL \sqcap Search = \bot$
$Search \sqsubseteq \top$	$GraphType \sqcap Search = \bot$
$\ldots$	$\ldots$

2. The root concept and features in a feature diagram are inter-related by various feature relations, represented by different edge types in the diagram. In our OWL model, for each of these edges, we create an object property. We assert that the range of the property is the respective feature class.

   $\top \sqsubseteq \forall\, hasG.G$ $\qquad$ $\top \sqsubseteq \forall\, hasF_i.F_i$, for $1 \leq i \leq n$

   For the GPL case study, we have one object property for each of the concept/features.

$\top \sqsubseteq \forall\, hasGPL.GPL$	$\top \sqsubseteq \forall\, hasSearch.Search$
$\top \sqsubseteq \forall\, hasBFS.BFS$	$\top \sqsubseteq \forall\, hasAlgorithm.Algorithm$
$\ldots$	$\ldots$

3. For each concept/feature node in the diagram, we create a *Rule* class. For each of these *Rule* classes, we add a necessary and sufficient (NS, `EquivalentClass`) condition, using an existential restriction, to bind the `Rule` node to the corresponding feature node in the diagram.

$GRule \sqsubseteq \top$	$F_iRule \sqsubseteq \top$, for $1 \leq i \leq n$
$GRule \equiv \exists\, hasG.G$	$F_iRule \equiv \exists\, hasF_i.F_i$, for $1 \leq i \leq n$

   For the GPL case study, part of the initial modeling is shown below.

$GPL \sqsubseteq \top$	$GraphType \sqsubseteq \top$
$GPLRule \sqsubseteq \top$	$GraphTypeRule \sqsubseteq \top$
$GPLRule \equiv \exists\, hasGPL.GPL$	$GraphTypeRule \equiv \exists\, hasGraphType.GraphType$
$GPL \sqcap GraphType = \bot$	$\ldots$

Now we are ready to model the feature relations. The general definition of each of the four feature relations will be shown, based on the above feature ontology. The GPL case study presented in Section 2 will be used to illustrate the idea. The ontology will be constructed incrementally to show the modeling of various feature relations and additional constraints defined in Table 2. As we will see, more axioms will be introduced to model the feature relations.

It is to be noted that the modeling of various feature relations and constraints will heavily rely on the *Rule* classes as well as the original feature classes. It is the *Rule* classes where all the OWL restrictions will be imposed.

Mandatory

A *mandatory* feature is included if its parent feature is included.

For each of the *mandatory* features $F_1, ..., F_n$ of a parent feature G, we use one N constraint to model it. It is a `someValuesFrom` restriction on $hasF_i$, stating that

each instance of the class *GRule* must have some instance of F_i class for $hasF_i$. The following ontology fragment shows the modeling of mandatory feature set and parent feature *G*.

$$GRule \sqsubseteq \exists\, hasF_1.F_1$$
$$\ldots$$
$$GRule \sqsubseteq \exists\, hasF_n.F_n$$

It can be seen from Fig. 1 that the root node *GPL* has a mandatory child feature *GraphType*, which is itself a non-leaf node. We create two new classes for these two non-leaf nodes[2].

$$GraphType \sqsubseteq \top$$
$$\top \sqsubseteq \forall\, hasGraphType.GraphType$$
$$GPLRule \sqsubseteq \exists\, hasGraphType.GraphType$$

The statement $\top \sqsubseteq \forall\, hasGraphType.GraphType$ is used to ensure the range of property *hasGraphType* to be *GraphType*. The statement $GPLRule \sqsubseteq \exists\, hasGraphType.GraphType$ ensures that GPL will have some GraphType as one of its child features.

Optional

An *optional* feature may or may not be included in a diagram, if its parent is included.

For each of the *optional* features $F_1, ..., F_n$ of a parent feature *G*, we need to ensure that if any of the feature from this set is included in a configuration, the parent feature also must be included. We accomplish this by making the Rule class of each child feature class a sub class of the `someValuesFrom` restriction on the parent feature (or concept) class.

$$F_1Rule \sqsubseteq \exists\, hasG.G \qquad \cdots \qquad F_nRule \sqsubseteq \exists\, hasG.G$$

In the GPL case study, *Search* is an optional feature for *GPL*. We model it as follows.

$$SearchRule \sqsubseteq \exists\, hasGPL.GPL$$

Alternative

As stated in Section 2, one and only one feature from a set of *alternative* features can be included, if their parent feature is included in a configuration.

For a set of *alternative* features $F_1, ..., F_n$ and a parent feature *G*, we use disjunction of `someValuesFrom` restrictions over $hasF_i$s to ensure that some feature will be included. We use the complement of conjunction of `someValuesFrom` restrictions to ensure that only one feature can be included. The symbols $\bigsqcup$ and $\bigsqcap$ represent distributed disjunction and conjunction respectively.

$$GRule \sqsubseteq \bigsqcup\, (\exists\, hasF_i.F_i), \text{ for } 1 \leq i \leq n$$
$$GRule \sqsubseteq \neg \bigsqcap\, (\exists\, hasF_i.F_i), \text{ for } 1 \leq i \leq n$$

[2] For brevity reasons, class definitions, disjointness and range statements will not be shown from here onwards.

Fig. 1 shows that features *BFS* and *DFS* compose an alternative feature set for *Search*. We model this relation as follows.

$$SearchRule \sqsubseteq (\exists\, hasBFS.BFS) \sqcup (\exists\, hasDFS.DFS)$$
$$SearchRule \sqsubseteq \neg\, ((\exists\, hasBFS.BFS) \sqcap (\exists\, hasDFS.DFS))$$

Or

According to Section 2, at least one from a set of *or* features is included, if the parent feature is included.

For a set of *or* features $F_1, ..., F_n$ of a parent feature G, we need to use a disjunction of `someValuesFrom` restrictions to model this relation.

$$GRule \sqsubseteq \bigsqcup\, (hasF_i.F_i), \text{ for } 1\ \leq i \leq n$$

It may be noticed that the definition of *or* is very similar to that of *alternative*, with the omission of the negation of conjunction to allow for multiple *or* features to be included.

In Fig. 1, the feature *Algorithms* has some *or* features. We use the following constructs to model it.

$$\begin{aligned} AlgorithmsRule \sqsubseteq\ & ((\exists\, hasNumber.Number) \sqcup \\ & (\exists\, hasConnected.Connected)\ \sqcup (\exists\, hasCycle.Cycle) \sqcup \\ & (\exists\, hasMST.MST)\ \sqcup (\exists\, hasShortest.Shortest) \sqcup \\ & (\exists\, hasStronglyConnected.StronglyConnected)) \end{aligned}$$

Requires

A feature may depend on some other features, hence its presence in a feature configuration *requires* the appearance of the others.

For a given feature G and a set of features $F_1, ..., F_n$ that *G requires*, we make sure that each of F_is appears in the configuration if G is present.

$$GRule \sqsubseteq \exists\, hasF_1.F_1$$
$$\ldots$$
$$GRule \sqsubseteq \exists\, hasF_n.F_n$$

In Table 2, feature *StronglyConnected* requires both *DFS* and *Directed*, and either *Weighted* or *Unweighted*. Its OWL representation is as follows.

$$StronglyConnectedRule \sqsubseteq \exists\, hasDFS.DFS$$
$$StronglyConnectedRule \sqsubseteq \exists\, hasDirected.Directed$$

Since *Weighted* and *Unweighted* form the set of two alternative features of *GraphType*, which is itself a mandatory feature and exactly one from a set of alternative features must appear in the configuration, we do not need to express them as additional constraints for *StronglyConnected*.

Excludes
The presence of a feature may be inhibited by that of some other features. We say the appearance of a feature in a configuration excludes the appearance of some other features.

For a given feature G and a set of features $F_1, ..., F_n$ that G excludes, we make sure, using the negation of `someValuesFrom` restriction on $hasF_i$ property, that G does not have any F_i feature .

$$GRule \sqsubseteq \neg\ (\exists\, hasF_1.F_1)$$
$$\ldots$$
$$GRule \sqsubseteq \neg\ (\exists\, hasF_n.F_n)$$

The next example shows both requires and excludes constraints for a single feature. In GPL, cycle checking algorithm *Cycle* excludes the use of breadth-first search *BFS*. From Table 2, we know that *Cycle* only requires *DFS*, hence it also excludes *BFS*.

$$CycleRule \sqsubseteq \exists\, hasDFS.DFS$$
$$CycleRule \sqsubseteq \neg\ (\exists\, hasBFS.BFS)$$

3.2 Feature Configuration Modeling

In feature modeling, a feature configuration derived from a feature model represents a concrete instance of a concept (i.e., a specific system in a domain). Intuitively, given a feature model ontology, features and concepts in a configuration should be instances (OWL individuals) of the classes defined in the ontology.

In our approach, we use OWL classes to simulate feature and concept instances so that the full power of the reasoning engines can be exploited to detect inconsistencies in the configuration.

Specifically, the reasoning support over ABoxes is not as comprehensive as that over TBox. ABox reasoners such as RACER can only detect that an ABox is incoherent w.r.t. a TBox *as a whole*. It cannot tell which instance(s) actually causes the problem. On the contrary, OWL reasoners can not only detect inconsistencies in a TBox, it can also show which class(es) are inconsistent.

Definition: feature configuration modeling
A feature configuration is a set of features that an instance of a concept may hold. The modeling of a given feature configuration is as follows.

- We model the concept node in the configuration as a subclass of the *Rule* class of the root concept in a feature diagram.
- We use an existential restriction for each feature included in the configuration.
- For each feature present in a feature diagram, we make its absence/presence explicit in a configuration according to this feature diagram to prevent the reasoning engine from erroneously inferring the existence of this feature in the configuration. This is necessary because of the Open World Assumption adopted by OWL.
- We make the concept class equivalent (NS condition) to the conjunction of the above constraints.

Without loss of generality, for a concept instance C derived from a feature diagram with root concept G and a set of features $F_1, ..., F_n$, assuming that $F_1, ..., F_i$ appear in the configuration of C and $F_{i+1}, ..., F_n$ do not, a feature configuration can be modeled as follows.

$$C \sqsubseteq GRule$$
$$C \equiv \bigsqcap (\exists\, hasF_j.F_j, \text{ for } 1 \leq j \leq i) \sqcap \bigsqcap (= 0\ hasF_k, \text{ for } i < k \leq n)$$

The feature configuration is constructed as a separate ontology and the reasoning engine is invoked to check its consistency. The configuration is valid if the ontology is checked to be consistent with respect to the feature diagram ontology.

3.3 Feature Configuration Verification in RACER

We use the GPL example to illustrate this approach. Suppose we have a configuration containing a concept instance E and some features for the GPL feature diagram in Fig. 1. We name the instance node the class E. Note that the namespace name of the feature diagram ontology is `GPL` and is omitted from the presentation.

$$\begin{aligned} E \sqsubseteq\ & GPLRule \\ E \equiv\ & ((\exists\, hasConnected.Connected) \sqcap (\exists\, hasSearch.Search) \sqcap \\ & (\exists\, hasAlgorithms.Algorithms) \sqcap (\exists\, hasBFS.BFS) \sqcap \\ & (\exists\, hasGraphType.GraphType) \sqcap (\exists\, hasNumber.Number) \sqcap \\ & (\exists\, hasWeighted.Weighted) \sqcap (\exists\, hasUndirected.Undirected) \sqcap \\ & (\exists\, hasStronglyConnected.StronglyConnected) \sqcap \\ & (= 0\ hasDirected) \sqcap (= 0\ hasMST) \sqcap (= 0\ hasShortest) \sqcap \\ & (= 0\ hasUnweighted) \sqcap (= 0\ hasDFS) \sqcap (= 0\ hasCycle)) \end{aligned}$$

If we input this ontology into Protégé and use RACER to *classify* the above ontology, RACER will complain that class E is inconsistent, as is shown in Fig. 2. A closer inspection reveals that *StronglyConnected* requires (via the Rule class) *DFS* and *Directed*, which are both absent in the configuration.

We correct the above configuration by asserting that E does have *DFS* and *Directed*. Since *BFS* and *DFS* and *Undirected* and *Directed* are alternative features, we remove *BFS* and *Undirected* from E.

$$\begin{aligned} E \sqsubseteq\ & GPLRule \\ E \equiv\ & ((\exists\, hasConnected.Connected) \sqcap (\exists\, hasSearch.Search) \sqcap \\ & (\exists\, hasAlgorithms.Algorithms) \sqcap (\exists\, hasDFS.DFS) \sqcap \\ & (\exists\, hasGraphType.GraphType) \sqcap (\exists\, hasNumber.Number) \sqcap \\ & (\exists\, hasWeighted.Weighted) \sqcap (\exists\, hasDirected.Directed) \sqcap \\ & (\exists\, hasStronglyConnected.StronglyConnected) \sqcap \\ & (= 0\ hasUndirected) \sqcap (= 0\ hasMST) \sqcap (= 0\ hasShortest) \sqcap \\ & (= 0\ hasUnweighted) \sqcap (= 0\ hasBFS) \sqcap (= 0\ hasCycle)) \end{aligned}$$

However, RACER again complains that concept E is inconsistent. The source of this inconsistency does not come from *StronglyConnected*. However, it is caused by the

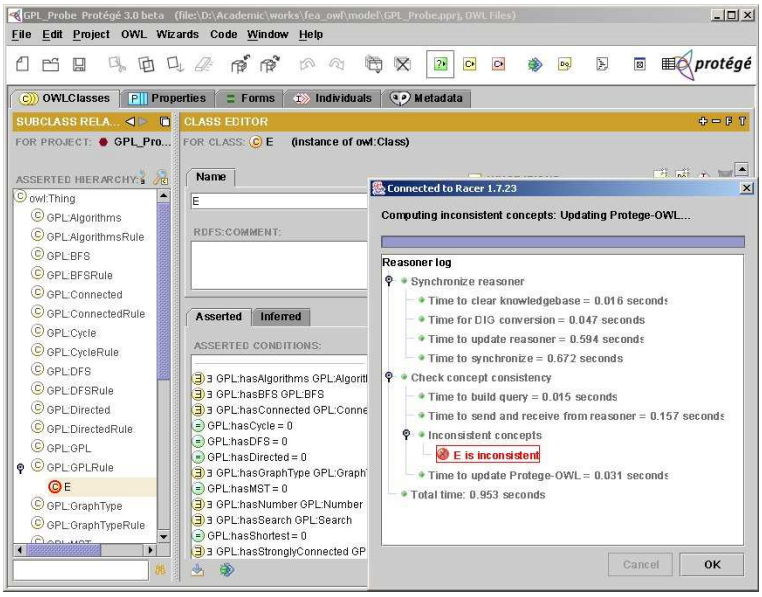


Fig. 2. RACER detects an inconsistency.

fact that feature *Connected* requires *Undirected*, which is absent from the configuration. Then we realize that features *StronglyConnected* and *Connected* are mutually exclusive in any valid configuration since they require different features from a set of *alternative* features.

After we remove *Connected* from the configuration of *E*, RACER confirms that the ontology is consistent, hence the configuration is valid.

Working on OWL DL, RACER can identify the inconsistency of a configuration with full automation. As pointed out in [5], with the growth of number of features in a feature diagram, manual checking of the consistency of a configuration is very laborious and highly error-prone. Since ontology reasoning tools are developed to reason about knowledge bases with enormous size, this approach is very scalable. The automated approach we adopt here is thus very advantageous.

4 A Tool Support for Feature Modeling in OWL

In the previous section, we present that OWL can be used to perform feature modeling. However it will be a tedious job for software engineers to design their system at such low level details. In this section we present a visual CASE tool that provides a high-level and intuitive environment for constructing feature models in OWL. Our feature modeling tool was built based on the meta-tool Pounamu [22]. Pounamu is a meta-case tool for developing multi-view visual environment. Fig. 3 shows the GPL feature model defined by the tool. From it we can see that the GPL feature model can be defined easily by creating instances of the pre-defined model entities and associations.

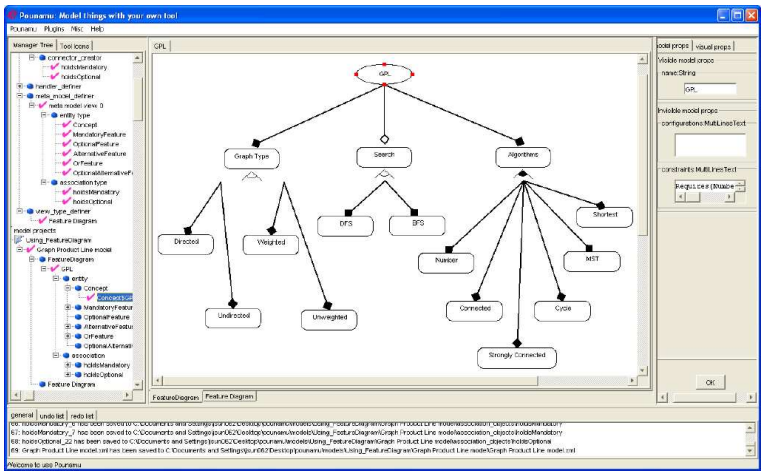


Fig. 3. A CASE tool for Feature Modeling, showing the GPL feature diagram.

Note that additional constraints among the features can also be specified in the "constraints" attribute of a concept. By triggering the defined event handler menu item in the tool, it transforms all the default XML format of each feature in the diagram into a single OWL representation of the feature model and save it for the RACER reasoning.

5 Conclusion

In this paper, we propose a Semantic Web environment to feature modeling and verification. We use the ontology language OWL DL to represent feature models and configurations in a formal and unambiguous way. By representing features as OWL classes and feature relations as OWL properties, the consistency of feature configurations can be automatically checked by Semantic Web reasoning engines such as RACER.

To facilitate visual development and analysis of feature models, we also developed a CASE tool that enables drawing feature diagrams and expressing additional constraints on various features. Feature diagrams are then converted to OWL syntax, made ready for online interchange and analysis. We used the Graph Product Line case study, a standard problem in the feature engineering community, throughout the paper to illustrate our approach.

There have been similar works on applying description logics to configuration problems [17, 7] and UML diagrams [1]. Compared to the description logics works, our approach has the following advantages.

– Compared to description logics, ontology languages and tools define mechanisms for management of ontologies. Hence, using ontology language for feature modeling makes the tasks of integrating, sharing, importing, versioning and maintaining feature models easy.

– Feature modeling languages can be enriched to support the modeling of features and feature relations of a higher complexity, beyond the expressivity of OWL. However, the expressiveness of the OWL language can be easily extended with the development of upper-layer languages such as the rule extension SWRL FOL [2]. With the development of tool support of these languages, more complex queries and checking can be performed for large-scale and complex feature models.
– The syntactic well-forcedness of OWL also makes it easy to transform feature ontologies to other useful formats for the purpose of visualization, concrete representation, etc.
– Being an open standard ontology language, potentially OWL has the advantage of ample reasoning support, which may not be restricted to description logics domain. For example, automated theorem provers (ATP) such as Vampire [20] have already been implemented to support reasoning OWL ontologies. Moreover, complex properties that cannot be handled by tools such as RACER may be solved by combined approaches such as [6].

We believe that Semantic Web can play important roles in domain engineering, and we will continue exploring the synergies between them. In the future, we plan to develop an integrated environment based on the current tool that supports the construction, analysis and exchange of the feature models in OWL.

Acknowledgement

The second author would like to thank Singapore Millennium Foundation (`http://www.smf-scholar.org/`) for the financial support. This work was supported in part by HyOntUse Project (GR/S44686) funded by the UK Engineering and Physical Science Research Council.

References

1. D. Berardi. Using dls to reason on uml class diagrams. In *Proc. of the 2002 Workshop on Applications of Description Logics (ADL 2002)*, Aachen, Germany, September 2002. CEUR (`http://ceur-ws.org/`.
2. Harold Boley, Mike Dean, Benjamin Grosof, Ian Horrocks, Peter Patel-Schneider, Said Tabet, and Gerd Wagner. SWRL FOL (November 2004). `http://www.daml.org/2004/11/fol/`, November 2004.
3. Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, MA., 2000.
4. D. Brickley and R.V. Guha (editors). Resource description framework (rdf) schema specification 1.0. `http://www.w3.org/TR/rdf-schema/`, February 2004.
5. Sybren Deelstra, Marco Sinnema, and Jan Bosch. Experiences in Software Product Families: Problems and Issues During Product Derivation. In Robert L. Nord, editor, *SPLC*, volume 3154 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 2004.
6. J. S. Dong, C. H. Lee, Y. F. Li, and H. Wang. A Combined Approach to Checking Web Ontologies. In *Proceedings of 13th World Wide Web Conference (WWW'04)*, pages 714–722, New York, USA, May 2004.

7. Michael Eisfeld. Model construction for configuration design. In *Proceedings of Workshop on Applications of Description Logics (ADL 2002)*, 2002.
8. M. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *The 5th International Conference on Software Reuse*, pages 76–85, Vancouver, BC, Canada, June 1998.
9. Volker Haarslev and Ralf Möller. Practical Reasoning in Racer with a Concrete Domain for Linear Inequations. In Ian Horrocks and Sergio Tessaris, editors, *Proceedings of the International Workshop on Description Logics (DL-2002)*, Toulouse, France, April 2002. CEUR-WS.
10. Volker Haarslev and Ralf Möller. *RACER User's Guide and Reference Manual: Version 1.7.6*, December 2002.
11. Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *J. of Web Semantics*, 1(1):7–26, 2003.
12. Kyo C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.
13. Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998.
14. Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 9:58–65, 2002.
15. Roberto E. Lopez-Herrejon and Don S. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, pages 10–24, Erfurt, Germany, September 2001. Springer-Verlag.
16. M. Dean and G. Schreiber (editors). OWL Web Ontology Language Reference. `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`, February 2004.
17. Deborah L. McGuinness. Configuration. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *Description Logic Handbook*, pages 388–405. Cambridge University Press, 2003.
18. M. Minsky. A framework for representing knowledge. In J. Haugeland, editor, *Mind Design: Philosophy, Psychology, Artificial Intelligence*, pages 95–128. MIT Press, Cambridge, MA, 1981.
19. Daniele Nardi and Ronald J. Brachman. An introduction to description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors, *The description logic handbook: theory, implementation, and applications*, pages 1–40. Cambridge University Press, 2003.
20. Alexandre Riazanov and Andrei Voronkov. Vampire. In Harald Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 292–296, Trento, Italy, July 7–10, 1999. Springer-Verlag.
21. F. van Harmelen, P. F. Patel-Schneider, and I. Horrocks (editors). Reference description of the DAML+OIL ontology markup language. Contributors: T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, L. A. Stein, et. al., March, 2001.
22. Nianping Zhu, John Grundy, and John Hosking. Pounamu: a meta-tool for multi-view visual language environment construction. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04)*, Rome, Italy, September 2004.