Verifying feature models using OWL

Hai H. Wang[a,*], Yuan Fang Li[b], Jing Sun[c], Hongyu Zhang[d], Jeff Pan[e]

[a] *University of Southampton, UK*
[b] *National University of Singapore, Singapore, Singapore*
[c] *The University of Auckland, Auckland, New Zealand*
[d] *Tsinghua University, China*
[e] *The University of Aberdeen, Aberdeen, UK*

Abstract

Feature models are widely used in domain engineering to capture common and variant features among systems in a particular domain. However, the lack of a formal semantics and reasoning support of feature models has hindered the development of this area. Industrial experiences also show that methods and tools that can support feature model analysis are badly appreciated. Such reasoning tool should be fully automated and efficient. At the same time, the reasoning tool should scale up well since it may need to handle hundreds or even thousands of features a that modern software systems may have. This paper presents an approach to modeling and verifying feature diagrams using Semantic Web OWL ontologies. We use OWL DL ontologies to precisely capture the inter-relationships among the features in a feature diagram. OWL reasoning engines such as FaCT++ are deployed to check for the inconsistencies of feature configurations fully automatically. Furthermore, a general OWL debugger has been developed to tackle the disadvantage of lacking debugging aids for the current OWL reasoner and to complement our verification approach. We also developed a CASE tool to facilitate visual development, interchange and reasoning of feature diagrams in the Semantic Web environment.

Keywords: Semantic Web; OWL; Ontologies; Feature modeling

1. Introduction

Domain engineering is a software reuse approach that focuses on a particular application domain. Examples of different domains are word processing, device driver for network adapters, 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 [1].

Feature modeling [2] 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” [2].

Quite a number of feature-based reuse approaches have been proposed, such as FODA (Feature-Oriented Domain Analysis) [3], FORM (Feature-Oriented Reuse Method) [4] and FeatuRSEB [5]. However, there is a lack of methods and tools that can support analysis over a feature model. Such methods and tools should provide us with a means of verifying the correctness of a feature model as the design of a feature model may be inconsistent. Once we have chosen a combination of features (a feature configuration) for a specific software product, such tools should be able to check the correctness of the configuration based on the constraints defined in the feature model. Furthermore, feature models may evolve when the knowledge of the domain increases. Thus when features are added/removed, such tools should enable us to check if a feature configuration is still valid. Industrial experiences show that in large scale software product line development, the number of features (variabilities)

[*] Corresponding author. Tel.: +44 2380593268; fax: +44 2380592865.
E-mail addresses: hw@ecs.soton.ac.uk (H.H. Wang), liyf@comp.nus.edu.sg (Y.F. Li), j.sun@cs.auckland.ac.nz (J. Sun), hongyu@tsinghua.edu.cn (H. Zhang), jpan@csd.abdn.ac.uk (J. Pan).

could be thousands and as a result, a substantial amount of effort is spent on correcting these human errors [6].

Due to the absence of a formal semantics of features and feature modeling, there is no mature tool that can check the correctness of a particular feature configuration based on the constraints specified in a feature model. Ideally, such a tool should bear a number of requirements:

Automated inconsistency detection. Different feature requirements may be contradictory and the product configuration may be invalid respecting to the feature model. In order to prevent inconsistent products being combined from incompatible features, it is important that inconsistencies can be detected automatically. It allows the domain experts to focus only on the system to be built, rather than the usability of the tool. Furthermore, the automation also enables the computer agents to compose software products run-timely based on users demands.

Reasoning efficiency. As a feature model may evolve constantly, specially for the dynamic re-configured feature systems, it requires the feature reasoning tool be able to conclude the validity of configurations in very short time.

Scalability. Modern software could be very large. Applications like Microsoft Windows OS have thousands of different features. The manual checking of such models/configurations are highly painstaking and error-prone. Hence, the feature reasoning system should scale up well to handle large and complex models.

Expressivity. As features interact with each other, the relationship among various features could be very complicated. The reasoning system should provide for means for representing and efficient reasoning over the wide variety of feature relations.

Debugging aids. It should provide some explanation as to why the feature models are inconsistent.

The Semantic Web has emerged as the next generation of the Web since the past few years. Ontology languages such as OWL [7] play a key role in realizing the full potential of the Semantic Web as they prescribe how data are defined and related. According to W3C, "an ontology defines the terms used to describe and represent an area of knowledge ... Ontologies include computer-usable definitions of basic concepts in the domain and the relationships among them ... They encode knowledge in a domain and also knowledge that spans domains. In this way, they make that knowledge reusable". One of the advantages of logic based ontology languages, such as OWL, in particular OWL-DL or OWL-Lite, is that reasoners can be used to compute subsumption relationships between classes and to identify unsatisfiable (inconsistent) classes. With the maturation of tableaux algorithm based DL reasoners, such as RACER [8], FaCT++ [9] and PELLET [10], it is possible to perform efficient reasoning on large ontologies formulated in expressive description logics.

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 them. Hence, we believe that the Semantic Web can play important roles in domain engineering.

In this paper, we explore the synergy of domain engineering and the Semantic Web. Given the rich expressiveness of OWL and its efficient and automated reasoning support, OWL can be adopted to reason and check feature models effectively. We propose methods for transforming a feature model into an OWL ontology. We use OWL reasoning engine such as FaCT++ [9] to perform automated analysis over an OWL representation of the feature model.

The analysis helps us detect possible inconsistencies in feature configurations. Furthermore, a general OWL debugger has been developed to tackle the disadvantage of lacking debugging aids for the current OWL reasoner and complement our verifying approach. We illustrate our approach using an example of the Graph Product Line (GPL) feature model, which is a standard problem proposed in [11] for evaluating product line technologies. Moreover, the performance of the approach has been evaluated by another very large and complicated feature model. Furthermore, we have developed a CASE tool to facilitate visual development, reasoning and distribution of Feature models in the OWL 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 describes the representation of a feature model in OWL. In Section 4, we show how FaCT++, a Semantic Web reasoning engine, can be used to perform automated analysis over the OWL representation of the feature models and present the OWL debugger to complement the FaCT++. In Section 5, we demonstrate the visual CASE tool we built to facilitate the creation and reasoning about feature models. Section 7 concludes the paper and describes future works.

2. Overview

2.1. *Feature modeling*

2.1.1. *Concepts and features*

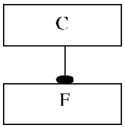
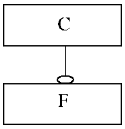
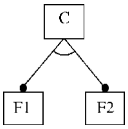
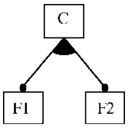
There are many definitions about features in the software engineering community, some of which are summarized below.

- A clustering of individual requirement that describe a cohesive, identifiable unit of functionality (Feature Engineering [12]).
- A prominent and distinctive user visible characteristic of a system (FODA [3]).
- A distinguishable characteristic of a concept that is relevant to some stakeholders (ODM [13]).

We use the ODM definition as it 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, we believe software features differentiate software systems. Features of a software system are not only related to user-visible functional requirements of the system, 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

Table 1
Types of features

Type	Notation
Mandatory	
Optional	
Alternative	
Or	

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.

2.1.2. Feature diagrams and feature relations

Conceptual relationships among features can be expressed by a feature model as proposed by Kang et al. [3].

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 types of features.

Table 1 provides an overview of some commonly found feature types. The graphical notation introduced by Czarnecki and Eisenecker [2] is 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, hence its presence is optional.
- *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.

Feature models are often used to model commonality and variability in a domain-engineering context. Commonalities can be modeled by common features (mandatory features whose ancestors are also mandatory), and variabilities can be modeled by variant features, such as optional, alternative, and or-features. A domain can be modeled as a concept.

Feature diagrams sometimes cannot capture all the interdependencies 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.

2.1.3. The graph product line feature model

The Graph Product Line (GPL) example is proposed by Lopez-Herrejon and Batory as a standard problem for evaluating software product line technologies [11]. 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 algorithm: 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 Fig. 1.

We also know from our knowledge of the graph algorithms that not all combinations of the features described in the above feature diagram (Fig. 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 it requires no search algorithm. Table 2 shows the additional constraints among the GPL features for representing a valid combination, adapted from Lopez-Herrejon and Batory [11].

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

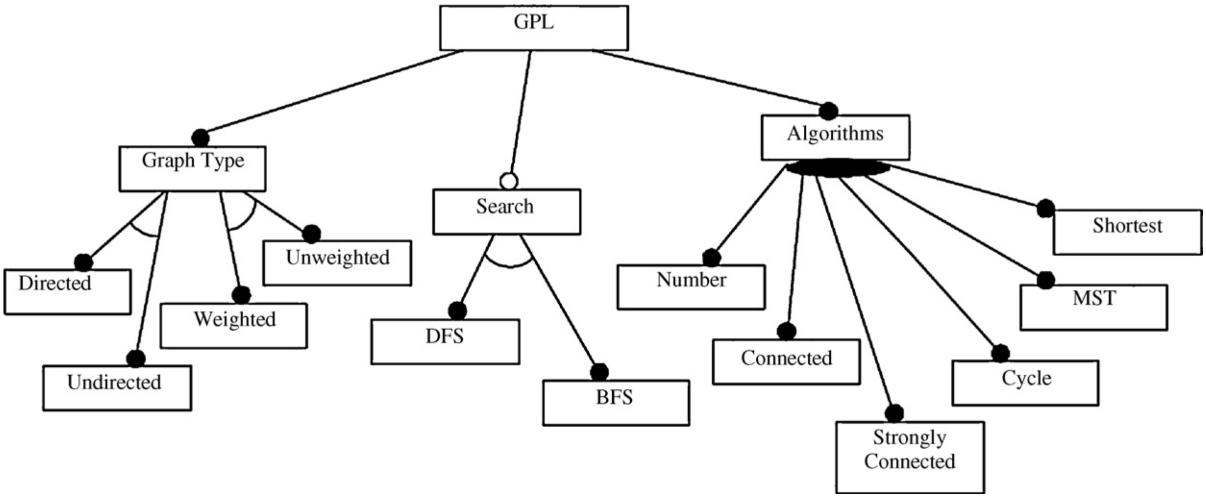


Fig. 1. A feature model for Graph Product Line.

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

From the above GPL model and additional constraints, we can see that (*GPL*, *GraphType*, *Directed*, *Unweighted*, *Algorithms*, *Number*) is a possible configuration derived from the *GPL* feature model. However, the configuration (*GPL*, *GraphType*, *Directed*, *Undirected*, *Weighted*, *Algorithms*, *Shortest*) is invalid since the features *Directed* and *Undirected* are exclusive to each other.

2.2. *The Semantic Web—languages and tools*

The Semantic Web was originally proposed by Berners-Lee et al. as “an extension to the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation” [14]. OWL is the latest standard in ontology languages, which was developed by members of the World Wide Web Consortium[2] and the DL community. An OWL ontology consists of classes, properties and individuals. Classes are interpreted as sets of objects that represent the individuals in the domain of discourse. Properties are binary relations that link individuals, and are interpreted as sets of tuples, which are subsets of the cross product of the objects in the domain of discourse.

Table 3 summarizes the “DL syntax” used in the following sections for feature modeling in OWL. Interested readers may refer to [15] for full details.

Ontology-related tools have been built alongside the development of ontology languages.

FaCT++ (**F**ast **C**lassification of **T**erminologies) [9] and RACER (**R**enamed **A**Box and **C**oncept **E**xpression **R**easoner [16]) are the two most widely accepted OWL reasoners. They support automated class subsumption and consistency reasoning and some queries on OWL ontologies.

Protégé [17] is a system for developing knowledge-based systems. It is an open-source, Java-based Semantic Web ontology editor that provides an extensible architecture, allowing users to create customized applications. In particular, the Protégé-OWL plugin [18] enables editing OWL ontologies and connecting to reasoning engines such as FaCT++ to perform tasks such as automated consistency checking and ontology classification.

Table 3
Summary of OWL syntax used in the paper

Notation	Explanation
$\top$	Super class of all OWL class
$N_1 \sqsubseteq N_2$	N_1 is a sub class/property of N_2
$\mathrm{C}_1 \sqsubseteq C\neg_2$	Classes C_1 and C_2 are disjoint
$C_1 \equiv C_2$	Class equivalence
$C_1 \sqcap / \sqcup C_2$	Class intersection/union
$\top \sqsubseteq \forall P \cdot C$	Range of property P is class C
$\forall / \exists P \cdot C$	allValuesFrom/someValuesFrom restriction, giving the class that for every instance of this class that has instances of property P, all/some of the values of the property are members of the class C

[2] http://www.w3.org.

To better present how to model and reason about feature models in OWL, we use Protégé-OWL as the ontology editor and FaCT++ as the reasoner. The OWL model will be automatically generated by our feature modeling CASE tool.

3. Feature modeling using OWL

In this section, we describe how to build a Semantic Web environment for feature modeling. Various feature relations can be model using OWL language constructs. We will discuss four types of feature relations: mandatory, optional, alternative, or and the two additional ones: requires and excludes. Modeling feature models using OWL has several advantages, such as facilitating feature model storing, sharing and distributing and assisting cooperative designing. In this paper, we only focus on verifying feature model using OWL.

In order to make use of the full power of FaCT++, we model the feature relations in OWL DL, since it is more expressive than OWL Lite and still retains decidability. As TBox reasoning is more comprehensive than that of ABox, we will model feature diagrams and feature configurations using OWL classes and properties instead of individuals. In this way the reasoning power of the engine is exploited to detect inconsistencies. Ref. [19] gives some more detailed discussion about the difference between those two modelling flavors.

Our presentation of the OWL encoding will be divided into two parts. Firstly, we present how a feature diagram and additional constraints are modeled in OWL, and in the second part, the modeling of feature configurations are discussed.

3.1. Conceptual modeling

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 a number of steps.

Step 1. We identify the nodes (concepts and features) present in a feature diagram. Each node in the diagram is modeled as an OWL class. Moreover, we assert that these classes are mutually disjoint. In OWL, all classes are assumed to overlap unless it is otherwise stated that they are *disjoint* with each other using a *disjoint axiom*. By default, we assume that features with different names are distinct.

Step 2. For each of these nodes in the diagram, we create a `Rule` class. This `Rule` class has two kinds of conditions: firstly, a necessary and sufficient (NS, `EquivalentClass`) condition, using an existential restriction to bind the `Rule` node to the corresponding feature node in the diagram; and secondly, a number of (possibly 0) necessary (N, `subClassOf`) constraints later, serving two purposes:

- To specify how each of its child features are related to this node, capturing the various relations between features.
- To specify how this feature node is constrained by other features, in the form of requires and excludes as discussed in Section 2.

Step 3. 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.

For a parent feature G and its child features $F_1, \ldots, F_n$, the initial modeling above produces the following ontology.

$$
\begin{array}{ll}
G \sqsubseteq \top & \mathit{hasG} \sqsubseteq \mathit{ObjectProperty} \\
\mathit{GRule} \sqsubseteq \top & \top \sqsubseteq \forall\ \mathit{hasG} \cdot G \\
 & \mathit{GRule} \equiv \exists\ \mathit{hasG} \cdot G \\
 & \\
F_1 \sqsubseteq \top & \\
F_1\mathit{Rule} \sqsubseteq \top & \mathit{hasF}_1 \sqsubseteq \mathit{ObjectProperty} \\
\ldots & \top \sqsubseteq \mathit{hasF}_1 \cdot F_1 \\
F_n \sqsubseteq \top & F_1\mathit{Rule} \equiv \exists\ \mathit{hasF}_1 \cdot F_1 \\
F_n\mathit{Rule} \sqsubseteq \top & \ldots \\
 & \mathit{hasF}_n \sqsubseteq \mathit{ObjectProperty} \\
G \sqsubseteq \neg F_i, \quad \text{for } 1 \le i \le n & \top \sqsubseteq \mathit{hasF}_n \cdot F_n \\
 & F_n\mathit{Rule} \equiv \exists\ \mathit{hasF}_n \cdot F_n \\
F_i \sqsubseteq \neg F_j, \quad \text{for } 1 \le i, j \le n \wedge i \ne j &
\end{array}
$$

Now we are ready to model the feature relations using the ontology. The general definition of each of the four feature relations will be shown, based on the above feature ontology. The GPL example presented in Section 2.1.3 will be used to illustrate the idea. The ontology will be constructed incrementally to show the modeling of various feature relations and addition constraints defined in Table 2.

3.1.1. Mandatory

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

For each of the *mandatory* features $F_1, \ldots, F_n$ of a parent feature G, we use one N constraints in *GRule* to model it. It is a `someValuesFrom` restriction on hasF_i, stating that each instance of the rule class 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.

$$
\begin{array}{l}
\mathit{GRule} \sqsubseteq \exists\ \mathit{hasF}_1 \cdot F_1 \\
\ldots \\
\mathit{GRule} \sqsubseteq \exists\ \mathit{hasF}_n \cdot F_n
\end{array}
$$

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.[3]

$$
\begin{array}{ll}
\mathit{GPL} \sqsubseteq \top & \mathit{hasGPL} \sqsubseteq \mathit{ObjectProperty} \\
\mathit{GraphType} \sqsubseteq \top & \top \sqsubseteq \mathit{hasGPL} \cdot \mathit{GPL} \\
\mathit{GPLRule} \sqsubseteq \top & \mathit{GPLRule} \equiv \exists\ \mathit{hasGPL} \cdot \mathit{GPL}
\end{array}
$$

$$
\begin{array}{l}
\mathit{GraphTypeRule} \sqsubseteq \top \\
\mathit{hasGraphType} \sqsubseteq \mathit{ObjectProperty} \\
\top \sqsubseteq \mathit{hasGraphType} \cdot \mathit{GraphType} \\
\mathit{GraphTypeRule} \equiv \exists\ \mathit{hasGraphType} \cdot \mathit{GraphType} \\
\mathit{GPLRule} \sqsubseteq \exists\ \mathit{hasGraphType} \cdot \mathit{GraphType}
\end{array}
$$

The statement $\mathit{GPLRule} \sqsubseteq \exists\ \mathit{hasGraphType} \cdot \mathit{GraphType}$ ensures that GPL will have some GraphType as one of its child features.

3.1.2. Optional

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

[3] Disjointness and range statements will not be shown from here onwards.

For each of the *optional* features $F_1, \ldots, F_n$ of a parent feature G, no additional statements are required to model this relationship.

Fig. 1 shows that feature *Search* is an optional feature for *GPL*. That is, *Search* may, or may not be included in a configuration of *GPL*.

As usual, we create one new class *SearchRule* for *Search* since it is a non-leaf node, and one object-property *hasSearch*. The ontology is augmented as follows. Note that no new restriction on *GPL* is added.

$$\begin{array}{ll} \mathit{Search} \sqsubseteq \top & \mathit{hasSearch} \sqsubseteq \mathit{ObjectProperty} \\ \mathit{SearchRule} \sqsubseteq \top & \mathit{SearchRule} \equiv \exists\, \mathit{hasSearch} \cdot \mathit{Search} \end{array}$$

3.1.3. 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.

Hence, for a set of *alternative* features $F_1, \ldots, F_n$ 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 distributed disjunction of the conjunction of two `someValuesFrom` restrictions to ensures that only one feature can be included. The symbol ⊔ represents distributed disjunction.

$$\mathit{GRule} \sqsubseteq \sqcup\, (\exists\, \mathit{hasF_i} \cdot F_i), \quad \mathit{for}\ 1 \le i \le n$$
$$\mathit{GRule} \sqsubseteq \neg \sqcup\, (\exists\, \mathit{hasF_i} \cdot F_i \sqcap \exists\, \mathit{hasF_j} \cdot F_j), \quad \mathit{for}\ 1 \le i \le j \le n$$

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

$$\begin{array}{ll} \mathit{BFS} \sqsubseteq \top & \mathit{hasBFS} \sqsubseteq \mathit{ObjectProperty} \\ \mathit{BFSRule} \sqsubseteq \top & \mathit{BFSRule} \equiv \exists\, \mathit{hasBFS} \cdot \mathit{BFS} \end{array}$$

$$\begin{array}{ll} \mathit{DFS} \sqsubseteq \top & \mathit{hasDFS} \sqsubseteq \mathit{ObjectProperty} \\ \mathit{DFSRule} \sqsubseteq \top & \mathit{DFSRule} \equiv \exists\, \mathit{hasDFS} \cdot \mathit{DFS} \end{array}$$

$$\mathit{SearchRule} \sqsubseteq ((\exists\, \mathit{hasBFS} \cdot \mathit{BFS}) \sqcup (\exists\, \mathit{hasDFS} \cdot \mathit{DFS}))$$
$$\mathit{SearchRule} \sqsubseteq \neg\, ((\exists\, \mathit{hasBFS} \cdot \mathit{BFS}) \sqcap (\exists\, \mathit{hasDFS} \cdot \mathit{DFS}))$$

The last two restrictions ensure that one and only one feature from the set of alternative features can be included.

3.1.4. 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 $F_1, \ldots, F_n$ of a parent feature G, we need to use a disjunction of someValuesFrom restrictions to model this relation.

$$\mathit{GRule} \sqsubseteq \sqcup\, (\exists\, \mathit{hasF_i} \cdot F_i), \quad \mathit{for}\ 1 \le i \le n$$

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

In Fig. 1, the feature *Algorithms* has a number of *or* features. We use the following constructs to model it. To save space, the definitions of the various `Rule` classes will not be shown.

$$\begin{array}{l} \mathit{Algorithms} \sqsubseteq \top \\ \mathit{AlgorithmsRule} \sqsubseteq \top \\ \mathit{hasAlgorithms} \sqsubseteq \mathit{ObjectProperty} \\ \mathit{AlgorithmsRule} \equiv \exists\, \mathit{hasAlgorithms} \cdot \mathit{Algorithms} \end{array}$$

$$\begin{array}{l} \mathit{AlgorithmsRule} \sqsubseteq ((\exists\, \mathit{hasNumber} \cdot \mathit{Number}) \sqcup \\ \quad (\exists\, \mathit{hasConnected} \cdot \mathit{Connected}) \sqcup (\exists\, \mathit{hasCycle} \cdot \mathit{Cycle}) \sqcup \\ \quad (\exists\, \mathit{hasMST} \cdot \mathit{MST}) \sqcup (\exists\, \mathit{hasShortest} \cdot \mathit{Shortest}) \sqcup \\ \quad (\exists\, \mathit{hasStronglyConnected} \cdot \mathit{StronglyConnected})) \end{array}$$

3.1.5. 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 $F_1, \ldots, F_n$ that G *requires*, besides the NS condition that binds *GRule* to G, we make sure that each of the F_i features appears in a configuration if G is present.

$$\begin{array}{l} \mathit{GRule} \sqsubseteq \exists\, \mathit{hasF_1} \cdot F_1 \\ \ldots \\ \mathit{GRule} \sqsubseteq \exists\, \mathit{hasF_n} \cdot F_n \end{array}$$

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

$$\begin{array}{l} \mathit{StronglyConnectedRule} \sqsubseteq \top \\ \mathit{StronglyConnectedRule} \equiv \exists\, \mathit{hasStronglyConnected} \cdot \mathit{StronglyConnected} \end{array}$$

$$\begin{array}{l} \mathit{StronglyConnected} \sqsubseteq \exists\, \mathit{hasDFS} \cdot \mathit{DFS} \\ \mathit{StronglyConnected} \sqsubseteq \exists\, \mathit{hasDirected} \cdot \mathit{Directed} \end{array}$$

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*.

3.1.6. 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 $F_1, \ldots, F_n$ that G excludes, we make sure, using the negation of `someValues-From` restriction on *hasF*$_i$ property, that *Grule* does not have any F_i feature.

$$\begin{array}{l} \mathit{GRule} \sqsubseteq \neg\, (\exists\, \mathit{hasF_i} \cdot F_i) \\ \ldots \\ \mathit{GRule} \sqsubseteq \neg\, (\exists\, \mathit{hasF_n} \cdot F_n) \end{array}$$

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*.

$$\begin{array}{l} \mathit{CycleRule} \sqsubseteq \top \\ \mathit{CycleRule} \equiv \exists\, \mathit{hasCycle} \cdot \mathit{Cycle} \end{array}$$

$$\begin{array}{l} \mathit{CycleRule} \sqsubseteq \exists\, \mathit{hasDFS} \cdot \mathit{DFS} \\ \mathit{CycleRule} \sqsubseteq \neg (\exists\, \mathit{hasBFS} \cdot \mathit{BFS}) \end{array}$$

The complete GPL ontology model in XML syntax can be found at http://www.comp.nus.edu.sg/∼liyf/GPL.owl.

4. Verifying and debugging feature configuration in OWL

4.1. Verifying feature configuration

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 ontology, features and concepts in a configuration should be ground instances (OWL individuals) of the OWL classes defined in the ontology. Hence modeling feature configurations using individuals is a straightforward approach.

However, a number of reasons made us model feature configurations as OWL classes but not individuals.

- Firstly, since feature models and individual feature configurations both classify individual implementations, representing the specific configuration as a sub-concept is the most natural from a semantic viewpoint.
- Secondly, the reasoning support that we need is more readily available in TBox than in ABox. (1) Inconsistencies associated with an ABox may not be discovered by the TBox reasoning engine such as FaCT++. (2) If an ABox reasoner discovers that a particular feature configuration is inconsistent, it can only indicate that the entire ontology (ABox) is incoherent. It cannot determine, however, which instances actually cause the inconsistency. This greatly increases the difficulty in debugging the configurations. On the other hand, most of the reasoners are capable of locating the specific classes that are inconsistent.

As a result, in our approach, we use classes to simulate feature and concept instances so that the full power of the reasoning engine can be exploited to detect inconsistencies in the configuration.

Definition 1. 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 for the root in a feature diagram.

We use an existential restriction for each feature included in the configuration.

For each feature F present in a feature diagram but not in its configuration, we use a "$\neg\exists\, hasF \cdot F$" restriction to prevent the reasoning engine from inferring the existence of this feature in the configuration. This is necessary because of the Open World Assumption adopted by OWL [20].

We make the concept class to be equivalent (NS condition) to the conjunction of the above constraints.

For a concept instance C derived from a feature diagram with root concept G and a set of features $F_1, \ldots, F_n$, assuming that $F_1, \ldots, F_i$ appear in the configuration of C and $F_{i+1}, \ldots, F_n$ do not, a feature configuration can be modeled as follows.

$$\begin{array}{l}
C \sqsubseteq GRule\\
C \equiv \sqcap(\exists\, hasF_j \cdot F_j, \quad for\ 1 \le j \le i) \sqcap\\
\quad \sqcap(\neg\exists\, hasF_k \cdot F_k, \quad for\ i < k \le n)
\end{array}$$

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.

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

$$\begin{array}{l}
E \sqsubseteq GPLRule\\
E \equiv ((\exists\, hasConnected \cdot Connected) \sqcap (\exists\, hasSearch \cdot Search) \sqcap\\
\quad (\exists\, hasAlgorithms \cdot Algorithms) \sqcap (\exists\, hasBFS \cdot BFS) \sqcap\\
\quad (\exists\, hasGraphType \cdot GraphType) \sqcap (\exists\, hasNumber \cdot Number) \sqcap\\
\quad (\exists\, hasWeighted \cdot Weighted) \sqcap (\exists\, hasUndirected \cdot Undirected) \sqcap\\
\quad (\exists\, hasStronglyConnected \cdot StronglyConnected) \sqcap\\
\quad (\neg\exists\, hasDirected \cdot Directed) \sqcap (\neg\exists\, hasMST \cdot MST) \sqcap\\
\qquad (\neg\exists\, hasShortest \cdot Shortest) \sqcap\\
\quad (\neg\exists\, hasUnweighted \cdot Unweighted) \sqcap (\neg\exists\, hasDFS \cdot DFS) \sqcap\\
\qquad (\neg\exists\, hasCycle \cdot Cycle))
\end{array}$$

If we input this ontology into Protégé and use FaCT++ to check it, FaCT++ will complain that E is inconsistent (Fig. 2). In Protégé the inconsistent classes are marked as red. A closer inspection reveals that *StronglyConnected* requires *DFS* and *Directed*, which are both absent in the configuration.

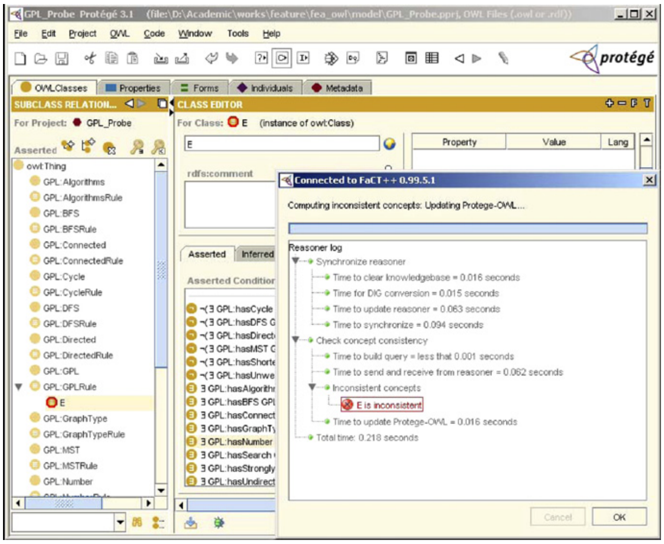
We correct the above configuration by asserting that E does have *DFS* and *Directed*. Since *BFS* and *DFS* and *Undirected* and *Directed* are alternative features, we remove *BFS* and *Undirected* from E.

$$\begin{array}{l}
E \sqsubseteq GPLRule\\
E \equiv ((\exists\, hasConnected \cdot Connected) \sqcap (\exists\, hasSearch \cdot Search) \sqcap\\
\quad (\exists\, hasAlgorithms \cdot Algorithms) \sqcap (\exists\, hasDFS \cdot DFS) \sqcap\\
\quad (\exists\, hasGraphType \cdot GraphType) \sqcap (\exists\, hasNumber \cdot Number) \sqcap\\
\quad (\exists\, hasWeighted \cdot Weighted) \sqcap (\exists\, hasDirected \cdot Directed) \sqcap\\
\quad (\exists\, hasStronglyConnected \cdot StronglyConnected) \sqcap\\
\quad (\exists\, hasUndirected \cdot Undirected) \sqcap (\exists\, hasMST \cdot MST) \sqcap\\
\qquad (\exists\, hasShortest \cdot Shortest) \sqcap\\
\quad (\exists\, hasUnweighted \cdot Unweighted) \sqcap (\exists\, hasBFS \cdot BFS) \sqcap (\exists\, hasCycle \cdot Cycle))
\end{array}$$

However, FaCT++ complains that the updated concept E is still inconsistent. The source of this inconsistency does not come from *StronglyConnected*. However, it is caused by the 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, FaCT++ confirms that the ontology is consistent, hence the configuration is valid.

Although FaCT++ cannot tell why a configuration is invalid (debugging feature models in OWL will be discussed in a later section), it can identify the inconsistency of a configuration with full automation. As the case study shows, with the growth of the number of features in a feature diagram, manual checking of the consistency of a configuration is very laborious and highly



```
Connected to FaCT++ 0.99.5.1

Computing inconsistent concepts: Updating Protege-OWL...

Reasoner log
▼ Synchronize reasoner
    Time to clear knowledgebase = 0.016 seconds
    Time for DIG conversion = 0.015 seconds
    Time to update reasoner = 0.063 seconds
    Time to synchronize = 0.094 seconds
▼ Check concept consistency
    Time to build query = less that 0.001 seconds
    Time to send and receive from reasoner = 0.062 seconds
  ▼ Inconsistent concepts
      E is inconsistent
    Time to update Protege-OWL = 0.016 seconds
  Total time: 0.218 seconds
```

Fig. 2. FaCT++ detects an inconsistency.

error-prone. Moreover, 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.

4.2. *Debugging feature OWL models*

The OWL reasoners, like FaCT++, can perform efficient reasoning on large ontologies automatically. Another important requirement for feature model reasoning tool, as discussed in Section 1 is the debugging support. However, the lack of debugging aids is a major shortcoming for the existing OWL reasoners. When checking satisfiability (consistency), the OWL reasoners can only provide a list of unsatisfiable classes and offer no further explanation for their unsatisfiability. It means that the reasoner can only conclude if a feature model is consistent and flag the invalid configurations. The process of "debugging" a feature model is left for the user. When faced with several unsatisfiable configurations in a moderately large feature model, even expert ontology engineers can find it difficult to work out the underlying error.

Debugging an ontology has been well recognized as a non-trivial task.

To provide some debugging assistance for the inconsistent feature models, we have built an OWL debugging tool based on the heuristics [21]. Our OWL debugger has been designed to adopt the general OWL DL ontology and it can be also used to explain the errors in the feature models as well.

Over the past five years we have presented a series of tutorials, workshops and post-graduate modules on OWL DL and its predecessors. Based on our experiences, a list of frequently made errors have been identified as reported in [22]. This catalogue of common errors has been used in turn to develop a set of heuristics that have been incorporated into a debugging tool for Protege-OWL [23].

The heuristic debugger treats the tableaux reasoner as a "black box" or "oracle". This "black box" approach has the advantage that it is independent of the particular reasoner used. It works with any DIG [24] compliant reasoner, even ones which have been specially augmented or adapted.

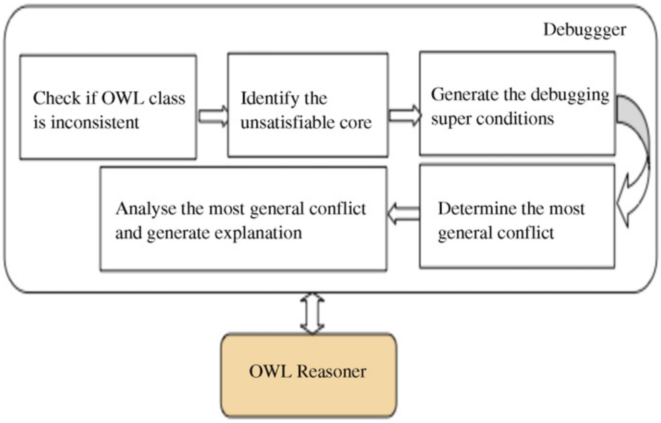

Debuggger
Check if OWL class is inconsistent
Identify the unsatisfiable core
Generate the debugging super conditions
Determine the most general conflict
Analyse the most general conflict and generate explanation
OWL Reasoner

Fig. 3. The debugging process.

Being independent of the reasoner has advantages even if only a single reasoner is to be used. Many modern reasoners transform the input ontology in order to optimize the reasoning process. Although logically equivalent, the internal representation may bear little resemblance to the ontology as it was constructed by the user. Given such transformations, even it were possible for the reasoner to 'explain' its actions, the explanation in terms of the transformed ontology would be unlikely to be of direct use to the user. An additional advantage of the 'black box' approach is that it is independent of such transformations.

4.2.1 Debugging process

Fig. 3 illustrates the main steps of the debugging process. The user selects an OWL class for debugging, which is checked to ensure it is indeed inconsistent, and that the user is making a valid request to the debugger. The debugger then attempts to identify the *unsatisfiable core* for the input class in order to minimize the search space. The *unsatisfiable core* is the smallest set of local conditions (direct super classes) that leads to the class in question being inconsistent. Having determined the unsatisfiable core, the debugger attempts to generate the *debugging super conditions*, which are the conditions that are implied by the conditions in the *unsatisfiable core*. Fig. 8 presents the rules that are used in generating the *debugging super conditions*. The debugger then examines the *debugging super conditions* in order to identify the *most general conflicting* class set, which is analyzed to produce an explanation as to why the class in question is inconsistent.

There are many different ways in which the axioms in an ontology can lead to an inconsistency. However, in general, we have found that most inconsistencies can be boiled down into a small number of 'error patterns'. In summary the 'error patterns' for class inconsistency may be boiled down to the following reasons:

The inconsistency is from some local definition.

(1) Having both a class and its complement class as super conditions.

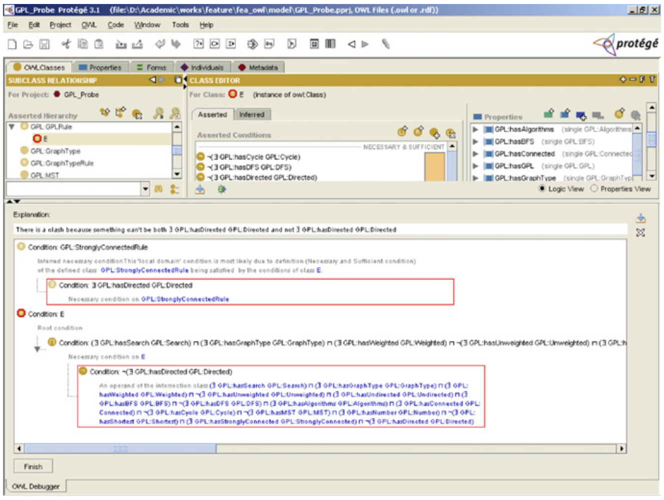
(2) Having both universal and existential restrictions that act along the same property, whilst the filler classes are disjoint.
(3) Having a super condition that is asserted to be disjoint with `owl:Thing` (⊤).
(4) Having a super condition that is an existential restriction that has a filler which is disjoint with the range of the restricted property.
(5) Having super conditions of *n* existential restrictions that act along a given property with disjoint fillers, whilst there is a super condition that imposes a maximum cardinality restriction or equality cardinality restriction along the property whose cardinality is less than *n*.
(6) Having super conditions containing conflicting cardinality restrictions.

The inconsistency is propagated from other source.

(1) Having a super condition that is an existential restriction that has an inconsistent filler.
(2) Having a super condition that is a hasValue restriction that has an individual that is asserted to be a member of an inconsistent class.

Due to the space limitation, some of the patterns have been omitted here. The debugger determines which of the above cases led to an inconsistency, and then uses provenance information that describes how the debugging super conditions were generated in order to determine the 'root' cause of the inconsistency.

Fig. 4 shows the result of debugging the GPL feature ontology. It suggests that the configuration *E* is invalid (The class *E* is inconsistent) because that *Directed* cannot both be present (∃ *hasDirected·Directed*) and absent (¬ ∃ *hasDirected·Directed*). *Directed* is explicitly stated to be absent. However, it is also required from feature *StronglyConnected*, which is present in the configuration (see Table 2 for details). As discussed before, there are more than one reasons leading the configuration *E* to being invalid. The debugger will pick one error each time. Note that the primal feedback from the debugger has been presented in the paper deliberately to give people an intuitive experience of the OWL debugger. Those information will be further processed for providing a nature explanation of the reason of a feature model been inconsistent and a configuration been invalid.


GPL_Probe Protégé 3.1 (file:\D:\Academic\works\feature\fea_owl\model\GPL_Probe.pprj, OWL Files (.owl or .rdf))
File Edit Project OWL Code Window Tools Help
protégé
OWLClasses Properties Forms Individuals Metadata
SUBCLASS RELATIONSHIP
CLASS EDITOR
For Project: GPL_Probe
For Class: E (instance of owl:Class)
Asserted Hierarchy
GPL:GPLRule
E
GPL:GraphType
GPL:GraphTypeRule
GPL:MST
Asserted Inferred
Asserted Conditions
NECESSARY & SUFFICIENT
¬(∃ GPL:hasCycle GPL:Cycle)
¬(∃ GPL:hasDFS GPL:DFS)
¬(∃ GPL:hasDirected GPL:Directed)
Properties
GPL:hasAlgorithms (single GPL:Algorithms
GPL:hasBFS (single GPL:BFS)
GPL:hasConnected (single GPL:Connected
GPL:hasGPL (single GPL:GPL)
GPL:hasGraphType (single GPL:GraphTyp
● Logic View ○ Properties View
Finish
OWL Debugger

Explanation:

There is a clash because something can't be both ∃ GPL:hasDirected GPL:Directed and not ∃ GPL:hasDirected GPL:Directed

Condition: GPL:StronglyConnectedRule
Inferred necessary condition This 'local domain' condition is most likely due to definition (Necessary and Sufficient condition) of the defined class GPL:StronglyConnectedRule being satisfied by the conditions of class E.

Condition: ∃ GPL:hasDirected GPL:Directed
Necessary condition on GPL:StronglyConnectedRule

Condition: E
Root condition

Condition: (∃ GPL:hasSearch GPL:Search) ⊓ (∃ GPL:hasGraphType GPL:GraphType) ⊓ (∃ GPL:hasWeighted GPL:Weighted) ⊓ ¬(∃ GPL:hasUnweighted GPL:Unweighted) ⊓ (∃ GPL:h
Necessary condition on E

Condition: ¬(∃ GPL:hasDirected GPL:Directed)
An operand of the intersection class (∃ GPL:hasSearch GPL:Search) ⊓ (∃ GPL:hasGraphType GPL:GraphType) ⊓ (∃ GPL: hasWeighted GPL:Weighted) ⊓ ¬(∃ GPL:hasUnweighted GPL:Unweighted) ⊓ (∃ GPL:hasUndirected GPL:Undirected) ⊓ (∃ GPL:hasBFS GPL:BFS) ⊓ ¬(∃ GPL:hasDFS GPL:DFS) ⊓ (∃ GPL:hasAlgorithms GPL:Algorithms) ⊓ (∃ GPL:hasConnected GPL: Connected) ⊓ ¬(∃ GPL:hasCycle GPL:Cycle) ⊓ ¬(∃ GPL:hasMST GPL:MST) ⊓ (∃ GPL:hasNumber GPL:Number) ⊓ ¬(∃ GPL: hasShortest GPL:Shortest) ⊓ (∃ GPL:hasStronglyConnected GPL:StronglyConnected) ⊓ ¬(∃ GPL:hasDirected GPL:Directed)

Fig. 4. Debugging GPL class.

4.3 Evaluation

To better evaluate our approach, we constructed a feature model for a large system. It contains around 1000 different features and more than 400 different feature relations covering *Mandatory*, *Alternative*, *Optional*, *Or* features and *Requires* and *Excludes* relations. Ten different configurations has been conducted respecting with the feature models. The evaluation was conducted on a Pentium IV 2.8 GHz system with 1 GB memory running Windows XP.

First, we transform the feature model into OWL (The transformation process is automatic). We then load the resulted ontology into FaCT++ and check its consistency.

Note that the feature configurations below are presented in the following syntax for brevity reasons. The symbol "+" represents set union. Hence, for example, `F416` + `F417` denotes the set of two members, `F416` and `F417`. The statement `Or(`PL, `F226`, `F416` + `F417)` denotes that with PL being the parent concept (or feature) node, features `F416` and `F417` form an `Or` feature set.

Protégé has been used for presenting the OWL ontology. FaCT++ concludes that the feature model is inconsistent using 28.238 s. A large portion of the time consumption is the overhead from the OWL editor Protégé itself, e.g. pre-possessing ontology and rendering the classes. The reasoning task itself only takes 5.306 s. The inconsistency is caused by `F136` and `F137` being alternative and they are both included in the description of PL. The detailed explanations are as follows.

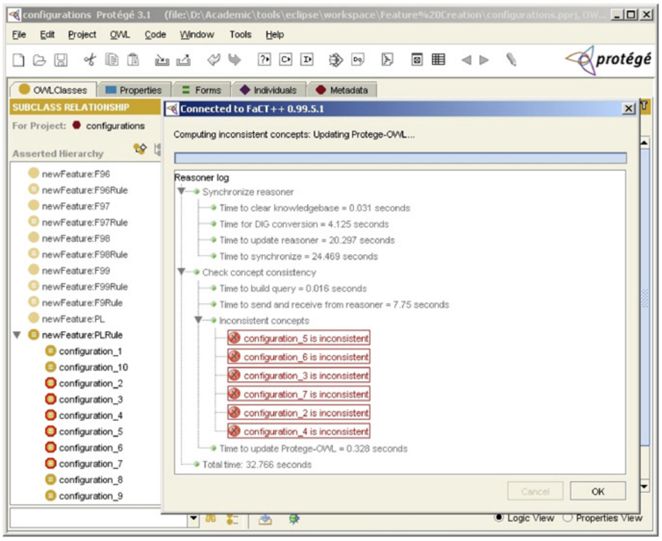
First of all `F6` is a *mandatory* feature of `PL`, `F126` is a required feature of `F6` and `F137` is a required feature of `F126`. Hence, `F137` must be held by `PL`.

In addition, `F416` is required by `F126`, so `F416` also must be held by `PL`. Because of the fact `Or` (`PL`, `F226`, `F416` + `F417`), since `F226` is the parent of `F416` and that `F416` is held by `PL`, according to the definition of *or* type, `F226` must be held by `PL`.

Because of the fact `Optional` (`PL`, `F136`, `F226` + `F227`) and the same reasoning as above, `F136` must also be held by `PL`. As `F136` and `F137` are alternative features, the model is inconsistent.

Our debugger can be used to trace these reasons effectively. If we remove the constraint that feature `F137` is a required feature of `F126`, the feature model becomes consistent, concluded by FaCT++ as well.

After that, we translate the 10 configurations into OWL, as shown in Fig. 5, FaCT++ picks up all the inconsistent configurations as expected. It takes only 32.766 s for FaCT++ to check the ten configurations. The reasoning task itself only takes 9.406 s. The debugger can be used to help us explain the reason why a configuration is invalid as well. For example, as shown in Fig. 6, the reason why the configuration 2 is invalid is because that `F416`, which is required by `F126` is missed from the configuration.


configurations Protégé 3.1 (file:\D:\Academic\tools\eclipse\workspace\Feature%20Creation\configurations.pprj, OW...
File Edit Project OWL Code Window Tools Help
protégé
OWLClasses Properties Forms Individuals Metadata
SUBCLASS RELATIONSHIP
For Project: configurations
Asserted Hierarchy
Connected to FaCT++ 0.99.5.1
Cancel
OK
Logic View
Properties View

newFeature:F96
newFeature:F96Rule
newFeature:F97
newFeature:F97Rule
newFeature:F98
newFeature:F98Rule
newFeature:F99
newFeature:F99Rule
newFeature:F9Rule
newFeature:PL
newFeature:PLRule
configuration_1
configuration_10
configuration_2
configuration_3
configuration_4
configuration_5
configuration_6
configuration_7
configuration_8
configuration_9

Computing inconsistent concepts: Updating Protege-OWL...

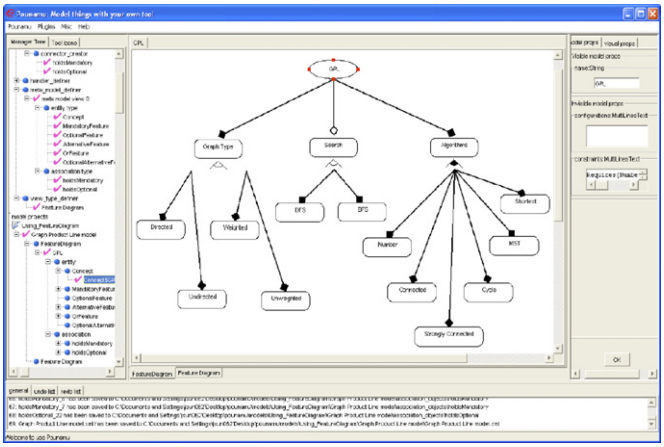
```
Reasoner log
▼ Synchronize reasoner
    Time to clear knowledgebase = 0.031 seconds
    Time for DIG conversion = 4.125 seconds
    Time to update reasoner = 20.297 seconds
    Time to synchronize = 24.469 seconds
▼ Check concept consistency
    Time to build query = 0.016 seconds
    Time to send and receive from reasoner = 7.75 seconds
  ▼ Inconsistent concepts
        configuration_5 is inconsistent
        configuration_6 is inconsistent
        configuration_3 is inconsistent
        configuration_7 is inconsistent
        configuration_2 is inconsistent
        configuration_4 is inconsistent
    Time to update Protege-OWL = 0.328 seconds
  Total time: 32.766 seconds
```

Fig. 5. Discovery of invalid configurations.

Apart from verifying if a configuration is entailed from the feature models (consistent), FaCT++ can also support the checking of semantic equivalence of feature models. Two feature models can be semantically equivalent even though they have different appearances in diagram. By "semantically equivalent", we mean that all valid feature instances (configurations) derived from one feature model can also be derived from the other model, and vice versa. In OWL, we can convert this problem to a subsumption (subclass) reasoning task and use FaCT++ to test it. This can also be done with full automation.

5. Tool support for feature modeling in OWL

In the previous section, we showed that OWL can be used to do the feature modeling. However it will be a tedious job for software engineers to design their system at such a level of details. In this section we present a visual case tool which 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 [25]. Pounamu is a meta-case tool for developing multi-view visual environment. Fig. 7 shows



67: holdsMandatory_7 has been saved to C:\Documents and Settings\jsun062\Desktop\pounamu\models\Using_FeatureDiagram\Graph Product Line model\association_objects\holdsMandatory
68: holdsOptional_22 has been saved to C:\Documents and Settings\jsun062\Desktop\pounamu\models\Using_FeatureDiagram\Graph Product Line model\association_objects\holdsOptional
69: Graph Product Line model.xml has been saved to C:\Documents and Settings\jsun062\Desktop\pounamu\models\Using_FeatureDiagram\Graph Product Line model\Graph Product Line model.xml

Fig. 7. A case tool for feature modeling.

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.

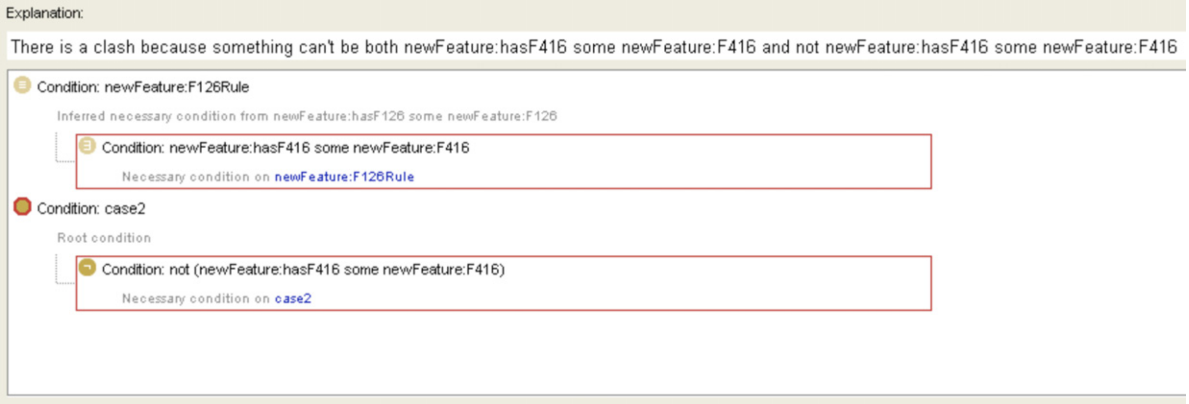
Note that additional constraints among the features can also be specified in the "constraints" attribute of a concept. By triggering the defined event handler menu item in the tool, it transforms all the default XML format of each feature in the diagram into a single OWL representation of the feature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

One undergoing development is to develop our tool as a plugin within the overall Protege plug-and-play framework.

6. OWL experiences discussion

In this paper, we presented an OWL application. We believe that feature modeling is a novel domain for OWL. In this section, we would like to feedback some of our experiences of using OWL to the Semantic Web community for the benefit of designing the next generation of OWL languages.

As shown in the previous sections, OWL provides a very expressive solution for providing fully automated, efficient and scalable reasoning service for verifying feature models. However, there are also some nuisances about current OWL.



Explanation:

There is a clash because something can't be both newFeature:hasF416 some newFeature:F416 and not newFeature:hasF416 some newFeature:F416

Condition: newFeature:F126Rule

Inferred necessary condition from newFeature:hasF126 some newFeature:F126

Condition: newFeature:hasF416 some newFeature:F416

Necessary condition on newFeature:F126Rule

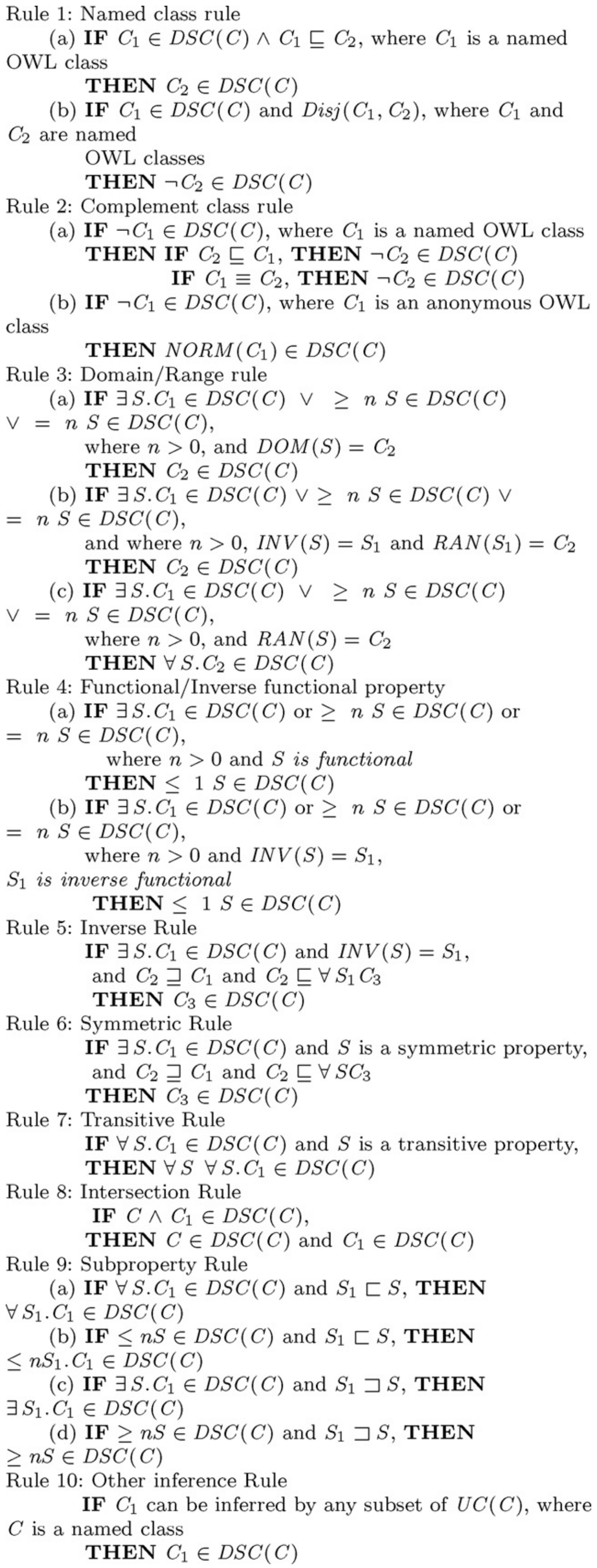
Condition: case2

Root condition

Condition: not (newFeature:hasF416 some newFeature:F416)

Necessary condition on case2

Fig. 6. Debugger determines why Configuration 2 is invalid.



```
Rule 1: Named class rule
    (a) IF $C_1 \in DSC(C) \wedge C_1 \sqsubseteq C_2$, where $C_1$ is a named
OWL class
        THEN $C_2 \in DSC(C)$
    (b) IF $C_1 \in DSC(C)$ and $Disj(C_1, C_2)$, where $C_1$ and
$C_2$ are named
        OWL classes
        THEN $\neg C_2 \in DSC(C)$
Rule 2: Complement class rule
    (a) IF $\neg C_1 \in DSC(C)$, where $C_1$ is a named OWL class
        THEN IF $C_2 \sqsubseteq C_1$, THEN $\neg C_2 \in DSC(C)$
                IF $C_1 \equiv C_2$, THEN $\neg C_2 \in DSC(C)$
    (b) IF $\neg C_1 \in DSC(C)$, where $C_1$ is an anonymous OWL
class
        THEN $NORM(C_1) \in DSC(C)$
Rule 3: Domain/Range rule
    (a) IF $\exists S.C_1 \in DSC(C) \ \vee \ \geq \ n\ S \in DSC(C)$
$\vee \ = \ n\ S \in DSC(C)$,
        where $n > 0$, and $DOM(S) = C_2$
        THEN $C_2 \in DSC(C)$
    (b) IF $\exists S.C_1 \in DSC(C) \vee \geq \ n\ S \in DSC(C) \vee$
$= \ n\ S \in DSC(C)$,
        and where $n > 0$, $INV(S) = S_1$ and $RAN(S_1) = C_2$
        THEN $C_2 \in DSC(C)$
    (c) IF $\exists S.C_1 \in DSC(C) \ \vee \ \geq \ n\ S \in DSC(C)$
$\vee \ = \ n\ S \in DSC(C)$,
        where $n > 0$, and $RAN(S) = C_2$
        THEN $\forall S.C_2 \in DSC(C)$
Rule 4: Functional/Inverse functional property
    (a) IF $\exists S.C_1 \in DSC(C)$ or $\geq \ n\ S \in DSC(C)$ or
$= \ n\ S \in DSC(C)$,
          where $n > 0$ and $S$ $is\ functional$
        THEN $\leq \ 1\ S \in DSC(C)$
    (b) IF $\exists S.C_1 \in DSC(C)$ or $\geq \ n\ S \in DSC(C)$ or
$= \ n\ S \in DSC(C)$,
        where $n > 0$ and $INV(S) = S_1$,
$S_1$ $is\ inverse\ functional$
         THEN $\leq \ 1\ S \in DSC(C)$
Rule 5: Inverse Rule
        IF $\exists S.C_1 \in DSC(C)$ and $INV(S) = S_1$,
         and $C_2 \sqsupseteq C_1$ and $C_2 \sqsubseteq \forall S_1 C_3$
         THEN $C_3 \in DSC(C)$
Rule 6: Symmetric Rule
        IF $\exists S.C_1 \in DSC(C)$ and $S$ is a symmetric property,
         and $C_2 \sqsupseteq C_1$ and $C_2 \sqsubseteq \forall SC_3$
        THEN $C_3 \in DSC(C)$
Rule 7: Transitive Rule
        IF $\forall S.C_1 \in DSC(C)$ and $S$ is a transitive property,
        THEN $\forall S \ \forall S.C_1 \in DSC(C)$
Rule 8: Intersection Rule
         IF $C \wedge C_1 \in DSC(C)$,
        THEN $C \in DSC(C)$ and $C_1 \in DSC(C)$
Rule 9: Subproperty Rule
    (a) IF $\forall S.C_1 \in DSC(C)$ and $S_1 \sqsubset S$, THEN
$\forall S_1.C_1 \in DSC(C)$
    (b) IF $\leq nS \in DSC(C)$ and $S_1 \sqsubset S$, THEN
$\leq nS_1.C_1 \in DSC(C)$
    (c) IF $\exists S.C_1 \in DSC(C)$ and $S_1 \sqsupset S$, THEN
$\exists S_1.C_1 \in DSC(C)$
    (d) IF $\geq nS \in DSC(C)$ and $S_1 \sqsupset S$, THEN
$\geq nS \in DSC(C)$
Rule 10: Other inference Rule
        IF $C_1$ can be inferred by any subset of $UC(C)$, where
$C$ is a named class
        THEN $C_1 \in DSC(C)$
```

Fig. 8. Rules for the membership of Debugging Super Conditions (DSC).

Apart from the lack for debugging aids as discussed before, another omission in the OWL language that we feel confounded is that there is no construct in OWL to make a set of classes mutually disjoint from each other, although OWL DL has the owl:AllDifferent construct to make a set of individuals mutually distinct from each other. This is because that the designers of OWL believe that "As the number of mutually disjoint classes grows the number of disjointness assertions grows proportionally to *n*2. However, in the use cases we have seen, *n* is typically small". The experiences in practice are that this is not the case—*n* is typically large enough such that the number of disjoint axioms becomes seriously problematic. For example, in our large feature models, there are about one thousand different features, which are distinct with each other. In the respected OWL ontology, almost 97% of axioms are the disjoint axioms for asserting the mutual disjointness. In terms of file size, these triples cause the 1 MB owl file to blow up to 24 MB. Similar problems are found in other biomedical ontologies such as GALEN, SNOMED, the NCI Thesaurus and the Gene Ontology. We believe that the owl:AllDisjoint construct should be added to the next version of OWL. Although it does not increase the expressive power, it will ameliorate the practical situation in terms of reducing model and file sizes, more efficient computation and clearer model content.

6.1. Feature modeling beyond OWL

OWL has considerable expressive power. All of the standard feature relationship types can be expressed within OWL. However, to retain the decidability of key inference problems in OWL DL and OWL Lite, OWL has expressivity limitations. Some non-standard and complicated feature type may not be able to be expressed within OWL. In this case, a more expressive language like SWRL [26] or SWRL-FOL [27] may be needed. However, currently there is no mature reasoning tools for these expressive languages.

7. Conclusion

In domain engineering, feature models are used to capture common and variant features among systems in a particular domain. Current efforts on feature modeling are largely graphical and informal, which have hindered precise representation and automated analysis of feature models. In the Semantic Web, an ontology is a representation of domain knowledge, which has formally-defined semantics and machine-understandable representation.

As both feature models and ontologies are intended to capture domain knowledge conceptually, it is natural to use ontology languages to rigorously represent and formally verify feature models and their configurations. The similarity between the two areas also suggests that ontology engineering techniques are applicable to feature modeling.

In this paper, we propose a Semantic Web approach to feature modeling, verification and debugging. We use the OWL DL language to represent feature models and configurations in an unambiguous way. Features of a particular domain are identified as OWL classes. Feature diagrams represent a graphical means of expressing relationships among different features. These relationships are captured by OWL properties. Feature configurations represent possible combinations (valid or invalid) of feature instances of a feature diagram. As discussed in Section 3, although it is natural to model feature configurations as OWL individuals, we model configurations using OWL classes in order to make use of the comprehensive reasoning support for TBox which is not available for ABox.

Feature model and configuration verification is an important task in feature modeling. With the growth of the number of features in a feature model, manual checking of validity is very laborious and error-prone. As OWL has a formal and rigorous semantical basis and the decidability of OWL DL, fully automated analysis is achievable.

In our approach, we use an OWL reasoning engines such as FaCT++ to perform automated analysis over the OWL representation of the feature models. The analysis helps us detect possible inconsistencies in feature configurations. As such reasoning engines are designed to handle large-scale knowledge bases, efficient and effective analysis of large feature models are possible.

The Graph Product Line example, a standard problem for evaluating software product line technologies, was used throughout the paper to illustrate our approach. We demonstrated that inconsistencies within various feature configurations are effectively detected by reasoning engines such as FaCT++.

Although reasoners such as FaCT++ and RACER are fully automated and very scalable. They cannot indicate the reasons as to why a class is inconsistent. With the growth of numbers of features in a feature diagram, the manual debugging of invalid configurations will be a very laborious and error-prone process. We use a general OWL debugger to automatically analyze an inconsistent concept instance. The debugger will provide some insight and hints on how the inconsistency is caused. This greatly helps to reduce the efforts and to improve debugging efficiency.

A large feature model containing some 1000 features with ten configurations was constructed to test the reasoning and debugging of feature models/configurations. It turns out that our approach is quite effective and precise.

To facilitate visual development and analysis of feature models, we also develop 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 believe that the 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 to support the construction, analysis and exchange of the feature models and configurations in OWL.

Acknowledgements

This work was supported in part by the EU-funded TAO project (IST-2004-026460), the HyOntUse Project (GR/S44686) funded by the UK Engineering and Physical Science Research Council and the Singapore Millennium Foundation.

References

[1] K.C. Kang, J. Lee, P. Donohoe, Feature-oriented product line engineering, IEEE Softw. 9 (2002) 58–65.

[2] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools and Applications, Addison-Wesley, MA, 2000.

[3] K.C. Kang, S. Cohen, J. Hess, W. Nowak, 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.

[4] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, FORM: a feature-oriented reuse method with domain-specific reference architectures, Ann. Softw. Eng. 5 (1998) 143–168.

[5] M. Griss, J. Favaro, M. d'Alessandro, Integrating feature modeling with the RSEB, in: Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, 1998, pp. 76–85.

[6] S. Deelstra, M. Sinnema, J. Bosch, Experiences in software product families: problems and issues during product derivation, in: R.L. Nord (Ed.), SPLC, Lecture Notes in Computer Science, vol. 3154, Springer, 2004, pp. 165–182.

[7] D.L. McGuinness, F. van Harmelen (Eds.), OWL Web Ontology Language Overview, 2003. http://www.w3.org/TR/2003/PR-owl-features-20031215/.

[8] V. Haarslev, R. Möller, RACER system description, in: Proceedings of the Automated Reasoning: First International Joint Conference, No. 2083 in Lecture Notes in Computer Science, Siena, Italy, 2001, pp. 701–706.

[9] I. Horrocks, Fact++ web site. http://owl.man.ac.uk/factplusplus/.

[10] B.P. Evren Sirin, Pellet: an owl dl reasoner, in: R.M. Volker Haaslev (Ed.), Proceedings of the International Workshop on Description Logics (DL2004), 2004.

[11] R.E. Lopez-Herrejon, D.S. Batory, A standard problem for evaluating product-line methodologies, in: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering, Springer-Verlag, Erfurt, Germany, 2001, pp. 10–24.

[12] C. Turner, A. Fuggetta, L. Lavazza, A. Wolf, A conceptual basis for feature engineering, J. Syst. Softw. 49 (1999) 3–15.

[13] M.S. et al., Software technology for adaptable reliable system (STARS) organization domain modeling (ODM) guidebook version 2.0, Tech. Rep. STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, Manassas, VA, 1996.

[14] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, Sci. Am. 284 (5) (2001) 35–43.

[15] I. Horrocks, P.F. Patel-Schneider, F. van Harmelen, From *SHIQ* and RDF to OWL: the making of a web ontology language, J. Web Semant. 1 (1) (2003) 7–26, URL download/2003/HoPH03a.pdf.

[16] V. Haarslev, R. Möller, RACER User's Guide and Reference Manual: Version 1.7.6, 2002.

[17] J. Gennari, M.A. Musen, R.W. Fergerson, W.E. Grosso, M. Crubezy, H. Eriksson, N.F. Noy, S.W. Tu, The evolution of protege: an environment for knowledge-based systems development, Tech. Rep. SMI-2002-0943, Stanford Medical Informatics, Stanford University, 2002.

[18] H. Knublauch, R.W. Fergerson, N.F. Noy, M.A. Musen, The Protégé OWL plugin: an open development environment for semantic web applications, in: Proceedings of the Third International Semantic Web Conference (ISWC 2004), Hiroshima, Japan, 2004.

[19] N. Noy, Representing classes as property values on the semantic web. http://www.w3.org/TR/2005/NOTE-swbp-classes-as-values-20050405.

[20] M.K. Smith, C. Welty, D.L. McGuinness (Eds.), OWL Web Ontology Language Guide, 2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[21] H. Wang, M. Horridge, A. Rector, N. Drummond, J. Seidenberg, Debugging OWL-DL Ontologies: A Heuristic Approach, in: Proceedings of the of Fourth International Semantic Web Conference (ISWC'05), Springer-Verlag, Galway, Ireland, 2005.

[22] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, C. Wroe, Owl pizzas: practical experience of teaching owl-dl: common errors and common patterns, in: Proceedings of the European Conference on Knowledge Acquistion, Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 63–81.

[23] A.R. Holger Knublauch, M. Musen, Editing description logic ontologies with the protégé-owl plugin International Workshop on Description Logics-DL2004, 2004.

[24] S. Bechhoffer, The dig description logic interface: Dig/1.1, Tech. rep., The University Of Manchester, The University Of Manchester, Oxford Road, Manchester M13 9PL, 2003.

[25] N. Zhu, J. Grundy, J. 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, 2004.

[26] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. http://www.w3.org/Submission/2004/SUBM-SWRL-20040521.

[27] H. Boley, M. Dean, B. Grosof, I. Horrocks, P. Patel-Schneider, S. Tabet, G. Wagner, SWRL FOL, 2004. http://www.daml.org/2004/11/fol/.