Feature Reasoning – Scaling Up and Pinning Down

Hai Wang
School of Computer Science
University of Manchester
Manchester, United Kingdom
hwang@cs.man.ac.uk

Yuan Fang Li
School Of Computing
National University of Singapore
Singapore
lyf@comp.nus.edu.sg

Jin Sun
Department of Computer Science
The University of Auckland
Auckland, New Zealand
j.sun@cs.auckland.ac.nz

Hongyu Zhang
School of Computer Science
and Information Technology
RMIT University
Melbourne, Australia
hongyu@cs.rmit.edu.au

Jeff Z. Pan
School of Computer Science
University of Manchester
Manchester, United Kingdom
pan@cs.man.ac.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. There is an increasing need for methods and tools that can support feature model analysis. Ideally, such reasoning tool should be fully automated, expressive and helpful in locating inconsistencies. At the same time, the reasoning tool should scale up well since it may need to handle hundreds or even thousands of features that modern software systems may have. In this paper, we propose to use existing software engineering techniques and tools, i.e., Z/EVES and Alloy Analyzer, complemented with novel Semantic Web ontology tools such as FaCT++ to check feature models. In this approach, FaCT++ is firstly used to identify any inconsistencies in a given feature model. Then the origins of the inconsistencies can be traced by Alloy Analyzer. Finally Z/EVES is used again to express complex non-standard feature relationship and to provide some advanced reasoning service. We have successfully applied this approach to a large and complicated feature model comprising 1000 features. A number of configurations have been checked and inconsistency have been detected and traced.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—Formal methods, Model checking; I.2.4 [Knowledge Representation Formalisms and Methods]: Representation languages

General Terms
Languages, Verification

Keywords
Feature modeling, formal methods, Z, Alloy, OWL, verification

1. INTRODUCTION
Research on features has received much attention in the software engineering community. Like objects and aspects, features are a dimension of concern that shall been considered in advanced “separation of concerns” methods [28]. Feature driven development has been proposed to develop software incrementally based on features [21]. In feature oriented programming [23], programs are created by composing features. In an application domain, a set of features gives rise to a software family. In generative programming [3] and software product line practices [23], domain analysis is performed to capture a product line’s commonalities and variabilities as features. Feature models [13, 14, 3] are proposed to model the structural relationship among features. A feature model contains a graphical tree-like notation that shows the hierarchical organization of features. Feature modeling is considered as “the greatest contribution of domain engineering to software engineering” [3]. According to Kang [15], customers and engineers usually speak of product characteristics in terms of the features the product has or delivers, so it is natural and intuitive to express any commonality or variability in terms of features.

Although much research has been centered on features, the concept of features and their relationships have not been well understood or formally defined. Many different kinds of notations (“languages”) have been proposed to assist feature modeling [13, 7, 3]. However, the lack of precision in the description of features and their relationships have prevented them from wider adoption. Nevertheless, there is an increasing need for methods and tools to support feature model analysis. 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 model 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) could be thousands and as a result, a substantial amount of effort is spent on correcting these human errors [4].

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. It should be possible to detect these inconsistencies in an automated manner.

**Debugging capability** It should provide some explanation as to why the feature models/configurations are inconsistent.

**Scalability** Modern software could be very large. The application 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 means for representing and efficient reasoning over the wide variety of feature relations.

However, as some of the requirements are very demanding, it is unlikely that in the near future such a theory and tool meeting all those requirements well can be developed. In this paper we propose to use novel technologies from the Semantic Web activity, i.e., the Web Ontology Language (OWL) [1] and its reasoners, complemented with the existing software engineering techniques and tools, i.e. Z and Alloy, to represent and reason over feature models and configurations.

The Semantic Web (SW) [2] provides a vision of having data on the web defined and linked in a way that they can be used for automation and integration. OWL is the standard SW ontology language, based on Description Logic (DL) [22]. With maturation of tableau algorithms based on DL reasoners such as FaCT [9], RACER [8] and FaCT++ [29], it is possible to perform efficient reasoning on large ontologies formulated in expressive description logics. The massive potential of SW technology in the context of software development has been foreseen by the Software Engineering Task Force (http://www.w3.org/2001/sw/BestPractices/SE/) in the W3C SW Best Practices and Deployment Working Group.

Z [31] is a widely used formal specification language designed to model system data and state. It is based on ZF set theory and first-order predicate logic, which can be regarded as a superset of description logic [16]. This intrinsic homogeneity implies that Z is more expressive than ontology languages and can capture complex constraints beyond current web ontology. Z/EVES [24] is an interactive proof tool for checking and reasoning Z specifications.

Alloy [10] was originally developed as a lightweight modeling language aimed at automated analysis. Its design was influenced by Z but is less expressive. Alloy Analyzer (AA) [11] is a fully-automated tool for analyzing Alloy specifications with special model checking capabilities which are helpful to trace the exact source of errors.

In our previous works [30, 27], we have applied OWL, Z-EVES/AA separately to formally represent and reason over feature models. RACER can detect inconsistencies in feature models and configurations highly efficiently and with full automation. AA can help to locate the source of errors. In Z/EVES, not only can we perform meta-level reasoning to ensure the correctness of the formal semantics, we can also define and reason over complex and non-standard feature relations, adding considerable expressive power to the feature modeling language. The above complementing properties make FaCT++ (or other OWL reasoners), Z/EVES and AA perfect ingredients for a combined approach, which we propose in this paper to tap their synergy.

We believe that the quality of feature model can be assured with higher confidence by using the synergy of reasoning power of OWL reasoners, Z/EVES and AA. In our approach, FaCT++ is first used to identify any inconsistencies in feature models/configurations. Then, AA is used as a surgery tool to trace the origins of such inconsistencies. Finally Z/EVES is used again to express complex and non-standard feature relations and reveal errors beyond the modeling capabilities of the current web ontology languages. By applying these tools systematically to a feature model/configuration, we can not only uncover more errors than using any one of them alone, but also correct any inconsistencies more easily and precisely.

The rest of the paper is organized as follows. We give an overview of Semantic Web, ontology languages, Z and Alloy and their tool support in Section 2. In Section 3, we briefly recall the semantics for feature modeling languages in Z, Alloy and OWL. Section 4 presents a detailed account of the combined approach, using a large feature model example that comprises 1000 features and some 400 various feature relations. Section 5 presents a prototype tool for developing, visualizing and reasoning over feature models and configurations. Finally, Section 6 discusses related works, future work directions and concludes the paper.

2. BACKGROUND INFORMATION

2.1 Feature Modeling - Preliminary

2.1.1 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.

2.1.2 Feature Diagrams & Feature Relations

Conceptual relationships among features can be expressed by a feature model as proposed in [13].

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:

---

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.

2.1.3 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 [17]. 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.

• Algorithms – A graph application implements one or more of the following algorithms: Vertex Numbering \((\text{Number})\), Connected Components \((\text{Connected})\), Strongly Connected Components \((\text{StronglyConnected})\), Cycle Checking \((\text{Cycle})\), Minimum Spanning Trees \((\text{MST})\), and Single-Source Shortest Path \((\text{Shortest})\).

• 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 \((\text{BFS})\) or Depth-First Search \((\text{DFS})\).

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

Table 1: Features types

<table>
<thead>
<tr>
<th>Type</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>Alternative</td>
<td>C</td>
</tr>
<tr>
<td>Or</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: A feature model for Graph Product Line.

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 [17].

Table 2: Additional Constraints on GPL

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Searches Required</th>
<th>Required Graph Type</th>
<th>Required Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Numbering</td>
<td>DFS, BFS</td>
<td>Directed, Undirected</td>
<td>Weighted, Unweighted</td>
</tr>
<tr>
<td>Connected Components</td>
<td>DFS, BFS</td>
<td>Undirected</td>
<td>Weighted, Unweighted</td>
</tr>
<tr>
<td>Strongly Connected</td>
<td>DFS</td>
<td>Directed</td>
<td>Weighted, Unweighted</td>
</tr>
<tr>
<td>Cycle Checking</td>
<td>DFS</td>
<td>Directed, Undirected</td>
<td>Weighted, Unweighted</td>
</tr>
<tr>
<td>Minimum Spanning Tree</td>
<td>None</td>
<td>Undirected</td>
<td>Weighted</td>
</tr>
<tr>
<td>Single-Source Shortest Path</td>
<td>None</td>
<td>Directed</td>
<td>Weighted</td>
</tr>
</tbody>
</table>

2.2 Semantic Web - Ontology languages & Tools

Description logics [20] are logical formalisms for representing information about classes, individuals and their relationships. It evolved from frame-based systems [19] and predicate logic and is well-known for the trade-off between expressivity and decidability.

Endorsed by W3C, OWL [18] 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 in full automation. 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.

FaCT++ [29] is the new generation of the well-known FaCT reasoner. It supports TBox (terminology box, classes) reasoning over OWL ontologies. Although it has yet provided the reasoning power over ABox (assertional box, individuals) and querying capabilities,
new optimizations have been introduced so that FaCT++ can perform core TBox reasoning tasks, such as concept consistency & subsumption checking and ontology classification much faster than other reasoners, such as RACER, for large ontologies.

2.3 Z & Alloy: Languages & Tools

Z [31] is a formalism based on ZF set theory and first-order predicate logic. It is specially suited to modeled system data and state. Z/EVES [24] is an interactive system for composing, checking, and analyzing Z specifications. It supports the analysis of Z specifications in a number of ways: syntax and type checking, schema expansion, precondition calculation, domain checking, general theorem proving, etc. In Z/EVES, Z specifications are in the form of sections to improve reuse. The built-in section toolkit defines basic constants and operators. Specifications are built hierarchically by including existing sections as their parents.

Alloy [10] is a structural modeling language emphasizing on automated reasoning support. It treats relations as first class citizens and uses relational composition as a powerful operator to combine various structural entities. The design of Alloy was influenced by Z and it can be (roughly) viewed as a subset of Z. Alloy Analyzer (AA) is a constraint solver that provides fully automated simulation and checking. AA works as a compiler: it compiles a given problem into a (usually huge) boolean formula, which is subsequently solved by a SAT solver, and the solution is then translated back to AA. Inevitably, a scope - a bound on the size of the domains - must be given to make the problem finite.

AA determines whether there exists a model of the formula. When AA finds an assertion to be false, it generates a counterexample, which (in some cases) makes tracing the error easier, compared to theorem provers. However, the capability of AA is constrained by the way it works. Since AA performs exhaustive search, it does not scale very well. In our experiences, AA can only handle specifications with no more than twenty entities. Similar to Z/EVES, Alloy specifications are in the form of modules, organized into a tree. Existing modules can be reused by commands open or use.

3. FEATURE MODELING USING Z, ALLOY & OWL

In this section, we present the formal semantics of feature modeling language in Z, Alloy and OWL. We firstly model the semantics in Z. As Z and Alloy are very similar, the Alloy semantics can be obtained from that of Z fairly straightforwardly. Secondly, based on the above formal semantics, we present the OWL semantics. As mentioned in Sec. 1, since OWL is less expressive than Z, the development of OWL semantics is less straightforward, which will become evident as we proceed. Thirdly, we demonstrate that by using Z/EVES, the semantics in Z (and hence Alloy & OWL) can be assured to capture what users intended.

3.1 Feature Modeling using Z/Alloy

In this section, we show how Z and Alloy can be used to formally represent concepts and relationships in feature modeling language.

3.1.1 Basic conceptual modeling

Features represent distinguishable characteristics of a concept. A concept consists of a set of related features with constraints. We give the definitions of Feature and Concept in Z as follows.

We define feature as a given set. Concept is a special kind of feature, which is represented as a subset of Feature.

\[ \forall c : \text{Concept} \rightarrow \text{Feature} \]

The above defines a relation holds that captures the relationship between a concept and each feature in the description of a concept instance. It represents a valid combination (configuration) of features that a concept instance can have. Each feature combination describes one possible instance of the concept. The predicate states that the concept node of a feature diagram is always included in any instance description derived from the diagram.

In Alloy, the following definitions define the above terms.

\[
\text{sig Feature} \\
\text{sig Concept extends Feature} \\
\]

3.1.2 Feature types modeling

Each of the feature types will be modeled in Z and Alloy as well. For example, Mandatory defines features that must be included into the description of a concept instance, if their parent feature is included. It can be defined formally in Z as follows.

\[ \forall c : \text{Concept} \rightarrow (\text{Feature} \times \mathbb{P} \text{Feature}) \]

The above defines Mandatory as a relation between a concept \( c \) and a pair of a parent feature \( pf \) and its direct child feature set \( s \). The first predicate states that the parent feature \( pf \) should not be included in the child set \( s \). The second and third predicates state that if the parent of the mandatory feature set \( s \) is held by a concept instance, all the features in \( s \) should be included into the description of the same concept instance; otherwise none. The Mandatory relation can be defined in Alloy as follows.

\[
\text{fun Mandatory(c:Concept,pf:Feature,s:set Feature)} \\
\text{(pf \: in \: c.holds \: \Rightarrow \: all \: f: \: s \: \mid \: f \: \: in \: c.holds \: pf \: \: in \: c.holds \: \Rightarrow \: all \: f: \: s \: \mid \: f \: \: !in \: c.holds \: \Rightarrow \: all \: f: \: s \: \mid \: f \: \: !in \: c.holds \: \Rightarrow \: all \: f: \: s \: \mid \: f \: \: !in \: c.holds)}
\]

Other feature relations can be modeled in Z and Alloy similarly.

3.1.3 Import Mechanisms & Proof Support

The Z semantics is contained in a section feature, on top of toolkit. Definitions alone are not sufficient to exploit the full power of Z/EVES. An ample stock of rewrite rules, forward rules and assumption rules is needed to make proof processes more automated. Based on the semantic model, we constructed a section, called feature\_theory, of rules which describes the above definitions in more than one angle and is used to help Z/EVES to perform reasoning tasks. This section has feature as parent. The Alloy semantics is contained in a module called FEATURE.

3.2 Feature Modeling using OWL

This subsection presents the representation of feature relationship types and feature models/configurations in OWL. As OWL is less expressive, the modeling is not so straightforward and structured as in the Z or Alloy approach and we do not have a semantic library in OWL. This signifies the trade-off between expressivity and decidability. Only part of the modeling is shown due to space limit, interested readers are referred to [30].
3.2.1 Feature Diagram Modeling

In this subsection, we present how a feature diagram and additional constraints are modeled in OWL. With feature models expressed in OWL, feature models can be verified using the OWL reasoner like FaCT++.

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 \( T \), the super class of every class). Moreover, we assert that these classes are mutually disjoint.

\[
G \subseteq T, \quad F_i \subseteq T, \\
G \cap F_i = \bot, \text{ for } 1 \leq i \leq n, \\
F_i \cap F_j = \bot, \text{ for } 1 \leq i < j \leq n
\]

Note that the clause \( G \cap F_i = \bot \) 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.

2. The 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.

\[
T \subseteq \forall \text{has}G.G \quad T \subseteq \forall \text{has}F_i.F_i, \text{ for } 1 \leq i \leq n
\]

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.

\[
\text{GRule} \subseteq T \\
GRule \equiv \exists \text{has}G.G \\
GRule \subseteq \forall \text{has}F_i.F_i, \text{ for } 1 \leq i \leq n
\]

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

\[
\text{GPL} \subseteq T \\
\text{GPLRule} \equiv \exists \text{hasGPL}.\text{GPL} \\
\text{GPL} \cap \text{GraphType} = \bot
\]

\[
\text{GraphType} \subseteq T \\
\text{GraphTypeRule} \equiv \exists \text{has}\text{GraphType}.\text{GraphType}
\]

Given the above basic model, the four types of feature relations can be modeled. The modeling of various feature relations and constraints heavily relies on the Rule classes as well as the original feature classes. It is the Rule classes where all the OWL restrictions will be imposed.

For example, 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 a N constraint to model it. It is a someValuesFrom restriction on \( \text{has}F_i \), stating that each instance of the class \( \text{GRule} \) must have some instance of \( F_i \) class for \( \text{has}F_i \). The following ontology fragment shows the modeling of mandatory feature set and parent feature \( G \).

\[
\text{GRule} \subseteq \exists \text{has}F_1, F_i, ... \\
\text{GRule} \subseteq \exists \text{has}F_n, F_n
\]

It can be seen from Fig. 1 that the root node \( GPL \) has a mandatory child feature \( \text{GraphType} \), which is itself a non-leaf node. We create two new classes for these two non-leaf nodes\(^1\).

\[
\text{GraphType} \subseteq T \\
T \equiv \forall \text{has}\text{GraphType}.\text{GraphType} \\
\text{GPLRule} \subseteq \exists \text{has}\text{GraphType}.\text{GraphType}
\]

The statement \( \text{GPLRule} \subseteq \exists \text{has}\text{GraphType}.\text{GraphType} \) ensures that GPL will have some GraphType as one of its child features.

Other feature relations like alternative, optional, or, requires, excludes can be similarly modeled.

3.2.2 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. Hence modeling feature configurations using individuals is a straightforward approach.

Since ABox reasoning is not as efficient as TBox reasoning, in our approach, we use classes to simulate feature and concept instances. More specifically, 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. A configuration is modeled 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.

\[
C \subseteq \text{GRule} \\
C \equiv \bigcap_{i} (\exists \text{has}F_j.F_j, \text{for } 1 \leq j \leq i) \cap \\
\bigcap_{k} (\text{has}F_k = 0, \text{for } i < k \leq n)
\]

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.

\(^1\)For brevity reasons, class definitions, disjointness and range statements will not be shown from here onwards.
3.3 Meta-level Reasoning Using Z/EVES

Because of the high expressivity of Z language, proof using Z/EVES is inevitably interactive. However, more advanced reasoning tasks can be performed with Z/EVES than with Semantic Web reasoners or Alloy. Meta-level reasoning over the Z semantics of feature relations can be performed to assure its correctness.

Although carefully developed, the formal semantics presented in the previous section may be erroneous, as it might not capture the meaning we intended. Hence, we need to prove its correctness, by stating desirable properties as theorems and prove them formally, using the Z/EVES theorem prover.

Moreover, definitions alone are not sufficient to prove properties about future feature models using Z/EVES efficiently and effectively. As suggested in [25], it is necessary to provide “a sufficient stock of theorems” to describe properties and sometimes simple facts of definitions and to express the relationship of several definitions so that proofs involving them can be well automated.

Hence, for the above two reasons we developed a library of theorems. To improve structural clarity and reuse, we made use of the section mechanism of Z/EVES. The formal semantics is put into a section feature_theory and the library of theorems is put in to a section feature_theory with section feature as its parent.

Below we show a fragment of the feature_theory section. Two simple theorems are shown. The first one, ConceptIsFeature, is an assumption rule; it states the fact that any concept is a feature. The second theorem is a rewrite rule, it states that \( (c, f) \in \text{holds} \) can be rewritten as \( f \in \text{holds}(\{c\}) \).

**Z Section feature_theory, parents: feature

**

\[ \text{theorem disabled grule ConceptsIsFeature} \]
\[ \forall c: \text{Concept} \quad c \in \text{Feature} \]

\[ \text{theorem rule imageHoldsRule} \]
\[ \forall c: \text{Concept, f: Feature} \quad (c, f) \in \text{holds} \iff f \in \text{holds}(\{c\}) \]

**end of Z Section feature_theory**

The library is incrementally and iteratively built such that whenever we encounter a theorem that is not easily proved, we observe the remaining goal and develop auxiliary theorems to facilitate the proof of current goal, after the auxiliary theorems are themselves proved. A few examples illustrate the theorem library and how proofs are constructed. From the semantics defined in Section 3.1, we know that if a set of features is defined as alternative and its parent feature is held by a concept, then exactly one member of the set can be held by that concept. In order to prove that the definition of alternative is correct, we constructed a number of theorems stating its various properties and proved them. For example, the following theorem, altTranRule3 states that the intersection of a set of alternative features \( s \) and the set of all the features held by the concept \( c \) is a singleton set containing \( g \), the one feature in \( s \) that is held by \( c \).

\[ \text{theorem altTranRule1} \]
\[ \forall c: \text{Concept, f: Feature, s: P_1 Feature} \quad (c, f) \in \text{holds} \land (c, (f,s)) \in \text{Alternative} \land g \in s \land (c, g) \in \text{holds} \Rightarrow s \cap \text{holds}(\{c\}) = \{g\} \]

\[ \text{theorem altTranRule2} \]
\[ \forall c: \text{Concept, f: Feature, s: P_1 Feature} \quad (c, f) \in \text{holds} \land (c, (f,s)) \in \text{Alternative} \Rightarrow \#(s \cap \text{holds}(\{c\})) = 1 \]

\[ \text{proof} \]
\[ \text{try lemma altTranRule2}; \]
\[ \text{simplify}; \]
\[ \text{use altTranRule1[c := c, f := f, s := s]}; \]
\[ \text{prove}; \]
\[ \text{equality substitute s \cap \text{holds}(\{c\}) = \{g\}}; \]
\[ \text{use sizeUnit[Feature][s := g]}; \]
\[ \text{reduce}; \]

The first one states that there indeed exists such a feature that is a member of the intersection of the two sets; and the second states that the cardinality of the intersection is actually 1. These two theorems are proved and the original theorem is proved based on them. The proof script of the second theorem is also shown. Clearly, it uses the result of theorem altTranRule1 to conclude that \( s \cap \text{holds}(\{c\}) \) is equivalent with \( \{g\} \), whose cardinality is 1.

Besides theorems about various definitions, there are also theorems about the relationship between various definitions in the library. For example, the following theorem involves Mandatory and Alternative feature types, stating that for two feature sets \( s_1, s_2 \) if they belong to Mandatory and the Alternative respectively for the same concept \( c \) under the same parent feature \( f \), then their intersection has only one element. Proof of this theorem involved another 4 auxiliary theorems, which describe some closely-related properties about the current theorem. For instance, one theorem states that given the above configuration, the intersection of the two sets \( s_1, s_2 \) is a subset of \( \text{holds}(\{f\}) \), the features held by \( c \).

\[ \text{theorem manAltRule3} \]
\[ \forall c: \text{Concept, f: Feature, s_1, s_2: P_1 Feature} \quad (c, f) \in \text{holds} \land (c, (f,s_1)) \in \text{Mandatory} \land s_1 \cap s_2 \neq \emptyset \land (c, (f,s_2)) \in \text{Alternative} \Rightarrow (\exists g: \text{Feature} \quad \{g\} = s_1 \cap s_2) \]

The theory library contains 49 theorems, all of which are formally proved by Z/EVES. These theorems serve two purposes:

- They help to prove the correctness the formal semantics by stating and proving desirable properties about definitions in the semantics. Proving the theorems gives us more confidence that the definitions in section feature capture the intended meaning.
• They can be used in proof of future, more complex feature models/configurations and relations. These theorems can help Z/EVES to assume facts and rewrite predicates to a closer form towards the goal. This will be shown in the next section.

4. COMBINED APPROACH TO CHECKING FEATURE MODEL

4.1 The Combined Approach

In this section, we present the approach of checking feature models/configurations using tools FaCT++, AA and Z/EVES in conjunction. Given a feature model or a configuration, we apply the tools in the following steps:

1. Firstly, we transform the feature model into an OWL ontology and use FaCT++ to classify it. In this step, FaCT++ will perform unsatisfiability checking which will decide whether there are inconsistencies in the feature model (whether the feature model could possibly have any configuration), and whether a configuration following a consistent feature model fully automatically. FaCT++ will report any inconsistent classes, however, it is unable to tell where the error lies and why the error occurs.

2. Secondly, in case of an error, we extract a small ontology fragment containing the relevant information about the inconsistent feature model and configuration. Then we employ AA to analyze the isolated information to determine the source of the error. In some cases, AA can pinpoint certain classes and properties which cause the error.

If the fragment model is too large for AA to analyze, we use Z/EVES as a theorem prover to determine the source of the inconsistency, which requires substantial expertise in interacting with Z/EVES.

After an error is reported, we check back the original model and correct it accordingly. FaCT++ is used again to check the consistency of corrected feature model and configuration.

3. Finally, we use Z/EVES again to check properties beyond the modeling capability of OWL DL and Alloy. As stated in section 2, Z is a superset of ontology languages and Alloy and it can capture a richer set of information, which is sometimes crucial to the correctness of the feature model.

4.2 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, Option, 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.

Firstly, we transform the feature model into OWL (the transformation process is automatic). We then load the resulted ontology into FaCT++ and classify it, as described in step 1 of Section 4.1. Protégé has been used for presenting the OWL ontology. In Protégé the inconsistent classes are marked as red. FaCT++ concludes that the feature model is inconsistent using 28.238 seconds. The large portion of the time consumption are the overhead from the OWL editor Protégé itself, e.g. pre-processing ontology and rendering the class. The reasoning task itself only takes 5.306 seconds. 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, F415 is required by F126, so F415 also must be held by PL. Because of the fact Or (PL, F226, F416+F417), since F226 is the parent of F415 and that F415 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. AA can be used to trace these reasons effectively (details will be shown in the next step). If we remove the constraint that feature F126 depends F137, the feature model become consistent, concluded by FaCT++ as well.

After that, we translate the 10 configurations into OWL, as shown in Figure 2, FaCT++ picks up all the inconsistent configurations as expected. It take only 32.766 seconds for FaCT++ to check the ten configurations. The reasoning task itself only take 9.406 seconds. Alloy can be used to help us to explain the reason a configuration is invalid as well.

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 subsumption (subclass) reasoning tasks and use FaCT++ to test it. This can also be done with full automation. Due to the space limitation, the details for this reasoning task are omitted from this paper.

FaCT++ can only tell us weather the feature model is consistent and which system configuration is valid. However, it cannot determine exactly where the inconsistency comes from. In the next step, we employ AA to pinpoint the source of the inconsistency.

Secondly, we extract a small ontology fragment containing the relevant information about the inconsistent model or invalid configuration [26]. These information is subsequently modeled into an Alloy module shown in Figure 3, which is loaded into AA to check for inconsistency. Please note that the order of the code has been manipulated for presentation purpose.

AA detects the inconsistency by its inability of finding a solution that satisfies all facts within the given scope. However, no solution may be due to the scope being too small. To determine the reason behind, we use AA’s utility “Determine unsat core” to trace the source of the error. If we are un-convinced of the error, we may increase the scope and run AA again.

Figure 4 shows how AA determines which facts caused the problem. In the right panel, clauses _Fact_144 to _Fact_147 are related to the problem. Arrows were added in the figure to show the correspondence of clauses in the right panel and concepts in the left panel. After examining the clauses, we found that the 4 clauses with arrows attached actually caused the problem. Hence, the lack of solution was indeed due to the inconsistency of the original ontology.

Thirdly, we will use Z/EVES to check for more complex feature relations. The feature modeling constructs we present in this paper are fairly comprehensive, however, they may need augmented to satisfy further requirements. The new feature types, however, may be easily represented and reasoned about in OWL or Alloy.

The Z language is a powerful modeling language that is capable of expressing arbitrarily complex formula in first-order predicate logic, making it an excellent candidate for further extending the
feature modeling language. In this subsection, we will present a small example to demonstrate the representation power and reasoning support of Z and Z/EVES in feature modeling.

Two possible complex feature relationships are holdsAtMostN and holdsAtLeastN, which state that for a concept \( c \), a parent feature \( pf \) and a set of features \( s \) to satisfy it, there can only be at most/at least \( n \) distinct features from \( s \) that are held by \( c \), if \( pf \) is held by \( c \), otherwise none.

Due to space limit, we will present a simplified version of holdsAtMostN in Z.

The above feature relationship is added to the feature model presented above. Before this addition, the configuration 10 was originally consistent. We want to make sure that with this additional feature type, this configuration is still consistent.
In the following we show the relevant part of this configuration 10 in Z. Note that statements about the distinctness of all the features are not shown.

```
PL : Concept
F113, F199, F200, F201 : Feature
...
(PL, F199) ∈ holds
(PL, F200) ∈ holds
(PL, F201) ∈ holds
...
```

Suppose that a new requirement requires that F113 and the set \{F199, F200, F201\} satisfy holdsAtMostN where \(n = 3\). We use the following theorem to state this.

**theorem testHamn**

\[ PL \, \text{HoldsAtMostN} \begin{cases} \text{pf}, \{F199, F200, F201\}, 3 \end{cases} \]

We enter the following commands in Z/EVES in hope that this theorem can be proved.

```
proof
  apply hammDef;
  reduce;
  use holds2;
  use holds3;
  apply imageHoldsRule;
  reduce;
  use testH[x := F199, y := F200, z := F201];
  reduce;
```

However, Z/EVES produces the following remaining goal:

\[
\#((\{F199\} \cup (\{F200\} \cup \{F201\})) = 3 \\
\land F199 \in \text{holds}(\{PL\}) \} \\
\land F200 \in \text{holds}(\{PL\}) \} \\
\Rightarrow \neg F201 \in \text{holds}(\{PL\}) \}
\]

Apparentlly this is a contradiction as we know that F201 is indeed held by PL. Hence, we negate the above theorem and try to prove it.

**theorem negTestHamn**

\[ \neg PL \, \text{HoldsAtMostN} \begin{cases} \text{F113}, \{F199, F200, F201\}, 2 \end{cases} \]

The above theorem can be formally proved to be true in Z/EVES using exactly the same proof commands as for theorem testHamn. Therefore, we know that the above features indeed do not satisfy holdsAtMostN relationship. The configuration hence is invalidated by the new requirement.

The above example demonstrates that Z and Z/EVES are indeed capable of capturing arbitrarily complex feature relationships and reasoning over configurations involving them.

## 5. TOOL SUPPORT FOR THE COMBINATION APPROACH

In the previous section, we present that OWL, Z and Alloy can be used for representing feature models. However, a case tool is desirable to assist the GUI design and the transformation for feature modeling. In this section we present a visual case tool which provides a high-level and intuitive environment for constructing feature models in OWL, Z and Alloy. Our feature modeling tool was built based on the meta-tool Pounamu [33]. Pounamu is a meta-case tool for developing multi-view visual environment. Fig. 5 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.

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 items in the tool, it transforms all the default XML format of each feature in the diagram into a single OWL, Z or Alloy representation of the feature model correspondingly and save them for the reasoning purposes.

## 6. CONCLUSION

Feature modeling has been widely used in software product line development and domain engineering. There is an increasing demand for methods and tools that support automated analysis of large-scale feature models [4]. In this paper, we propose a combined, scalable approach to feature model verification. Our approach utilizes the Semantic Web technologies (OWL and DL reasoners), complemented by the formal software engineering methods (Z and Alloy) and their proof tool support. We firstly propose formal semantics of feature modeling languages in Z, Alloy and OWL. We then transform a feature model into an OWL representation, and use a DL reasoner such as FaCT++ to check the consistency of the model and feature configurations. To help “debug” a feature model, we also use the Alloy Analyzer (AA) and Z/EVES to trace the origins of inconsistencies. We evaluated our approach using a feature model with 1000 nodes and some 400 feature relations. We believe our approach is novel and is useful for industrial practices.

Recently, some researchers have also begun to explore the formal semantics of feature models. For example, in [12], description logic is used for feature modeling. In [5], an algebra-based approach is proposed. However, their approaches do not provide automated tool support and their reasoning capacity is weaker than our combined approach. Batory proposed an approach [6] that represents a feature model using a “tree grammar” and maps the grammar to propositional formulas. Logic-maintenance systems and SAT solvers are then used to check the consistency of the feature model. The drawback of their approach is that the consistency of the feature model cannot be proved. Rather, test cases are run to obtain higher assurance without positive confirmation. Our combined ap-
proach, on the other hand, can not only verify the consistency of feature models/configurations with full automation, but also identify the source of the error if there is any inconsistency.

Research on feature interaction problems [32] in telecommunication domain concentrated on detecting behavioral interactions. Feature interaction occurs when one feature affects the operation of the other at system runtime. Our work concentrates on the structural relationships among features. We propose methods for automated consistency checking of features. Exploring dynamic interactions among features is an important future research area.

7. REFERENCES

