Formal Semantics and Verification for Feature Modeling

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

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

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

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

Abstract

Research on features has received much attention in the domain engineering community. Feature modeling plays an important role in the design and implementation of complex software systems. However, the presentation and analysis of feature models are still largely informal. There is also an increasing need for methods and tools that can support automated feature model analysis. This paper presents a formal engineering approach to the specification and verification of feature models. A formal semantics for the feature modeling language is defined using first-order logic. It provides a precise and rigorous formal interpretation for the graphical notation. In addition, further validation of the semantics using the Z/EVES theorem prover is presented. Finally, we demonstrate that the consistency of a feature model and its configurations can be automatically verified by encoding the semantics into the Alloy Analyzer. A case study of the Key Word in Context (KWIC) index systems feature model is presented to illustrate the verification process.

Keywords*: Feature Modeling, Domain Engineering, Feature Oriented Domain Analysis, Z/EVES, Alloy, Formal Verification.*

1 Introduction

Research on feature modeling has received much attention in the domain engineering community. In an application domain, a set of features gives rise to a software product line [14]. Feature-Oriented Reuse Method (FORM) [6] and Feature-Oriented Domain Analysis (FODA) [5] are domain engineering methods that concentrate on modeling and analyzing a product line's commonalties and variabilities in terms of features. According to Kang et al., customers and engineers usually speak of product characteristics in terms of the features that the product has or delivers; thus it is natural and intuitive to express any commonality or variability in terms of features [7]. FORM and FODA are known for the introduction of feature models, which contain 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" [1].

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 graphical notations ('languages') have been proposed to assist feature modeling [2, 1]. However, the lack of precision in the description of features and their relationships have prevented them from a wide adoption. In addition, there is an increasing need for methods and tools that can support automated feature model analysis. In the first paper on feature modeling in 1990 [5], Kang et al. suggested the formalization of features as a future direction and mentioned the possibility of applying an algebraic-based technique. However, since then formalization of feature models has not been taken up. In this paper, we present an approach to formalizing and verifying feature models using formal reasoning techniques. A first-order semantics for the feature modeling language is defined. Theorem proving techniques are used to validated the correctness of the semantics. Furthermore, we demonstrate that the consistency of a feature model and its configurations can be automatically verified, where the source of inconsistency can be identified.

The remainder of the paper is organized as follows. Section 2 gives a brief overview of feature modeling, Z, Z/EVES and Alloy. In Section 3, we present a formal semantics for the feature modeling language. Section 4 describes the meta-level reasoning support for validating the correctness of the semantics. In Section 5, we present a case study to demonstrate the feature model verification using the Alloy Analyzer. We use the Key Word in Context (KWIC) index systems as an example to illustrate the verification process. Section 6 concludes the paper and discusses the future works.

2 Background

2.1 Feature modeling

There are many definitions about features in the software engineering community, such as those found in Feature Engineering [13], FODA [6] and ODM [12]. We choose the ODM definition, "*Feature is a distinguishable characteristic of a concept that is relevant to some stakeholders*", 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 that software features differentiate software systems. 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.

Conceptual relationships among features can be expressed by a feature model as proposed by Kang et al. [5]. 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.

Kang et al. classified features as mandatory, optional and alternative features, and introduced an AND-OR graph based notation for representing feature models [5]. Many other researchers proposed different graphical notations and 'languages' for feature modeling. For example, Czarnecki and Eisenecker [1] proposed the or, optional-alternative, optional-or features and new notations; Griss et al. [2] proposed XOR and OR features. Many of these notations and 'languages' refer to same or similar relationships. There is a lack of commonly-accepted, precise definitions in feature modeling research.

Table 1 provides an overview of some commonly found feature types. We use the graphical notation introduced by Czarnecki and Eisenecker [1]. In Table 1, assuming the concept *C* is selected, we have the following definitions on its child features:

Type	Notation
Mandatory	C F
Optional	C F
Alternative	C F1 F2
Or	C F1 F2
Optional Alternative	C F1 F2
Optional Or	C F1 F2

Table 1. Types of features in a feature diagram

- 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.
- Optional Alternative – One or more features from a set of *Alternative* features is optional.
- Optional Or – One or more features from a set of *Or* features is optional.

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. Figure 1 shows a simple feature model for a *Car* domain [1].

From the *Car* feature model (Figure 1), we can see that *(Car, CarBody, Transmission, Manual, Engine, Gasoline)*

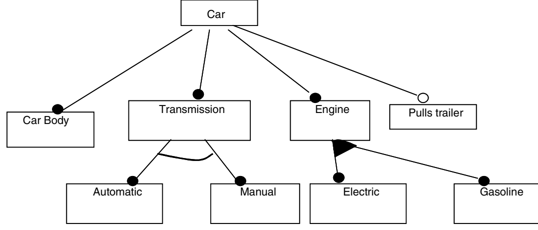

Figure 1. A simple Car feature model.

and *(Car, CarBody, Transmission, Automatic, Engine, Electric, Gasoline, PullsTrailer)* are possible configurations derived from the *Car* feature model. However, not all combinations of features are valid. For example, the configuration *(Car, CarBody, Transmission, Automatic, Manual, Engine, Gasoline)* is invalid since the features *Automatic* and *Manual* are exclusive to each other.

2.2 Z and Z/EVES

Z [15] is a state-based formal language based on ZF set theory and first-order predicate logic. Z/EVES [9] is an interactive system for composing, checking, and analyzing Z specifications. In particular, it supports general theorem proving of Z specifications. Z/EVES organizes Z specifications in the form of sections to improve structure and reuse. The built-in section `toolkit` defines basic constants and operators. Specifications are built hierarchically by including existing sections as their parents.

In Z/EVES, theorems can be developed to describe properties about a model. They appear in the form of axioms, rewrite rules or assumption rules. An axiom is treated as a fact, which is not invoked during reduction. When Z/EVES encounters a rewrite rule, it will rewrite its left-hand side to its right-hand side. An assumption rule is assumed to be true when Z/EVES performs simplification. Theorems can be marked as `disabled` so that they are not automatically invoked during reduction or rewriting.

2.3 Alloy

Alloy [3] is a structural modeling language based on first-order logic for expressing complex structural constraints and behaviors. It can be viewed as a subset of Z since it is less expressive. Alloy treats relations as first-class citizens and uses relational composition as a powerful operator to combine various structured entities. Like Z, Alloy is a declarative language. The essential constructs of Alloy are signatures, facts, functions and assertions.

The Alloy Analyzer [4] is a tool for analyzing models written in Alloy. Given a finite scope for a specification, Alloy Analyzer translates it into a propositional formula and uses SAT solving technology to generate instances that satisfy the properties expressed in the specification. It supports two kinds of automatic analysis: simulation, in which the consistency of an invariant or operation is demonstrated by generating a state transition; and checking, in which a consequence of the specification is tested by attempting to generate a counterexample.

3 Formal semantics for feature modeling

In this Section, we present a formal semantics for the feature modeling language using the first-order logic in Z. We show that all the commonly-used feature types in Section 2.2 can be expressed precisely. Furthermore, we define two more relations to capture the additional constraints among the features in a feature model.

3.1 Feature and concept

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* as follows.

$$[Feature] \qquad\qquad \mid Concept : \mathbb{P}\, Feature$$

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

$$\begin{array}{|l} holds : Concept \leftrightarrow Feature \\ \hline \forall c : Concept \bullet (c, c) \in holds \end{array}$$

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 above predicate states that the concept node of a feature diagram is always included in any instance description derived from the diagram.

3.2 Formal definitions of feature types

3.2.1 Mandatory

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 as follows.

$$\begin{array}{|l} Mandatory : Concept \leftrightarrow (Feature \times \mathbb{P}\, Feature) \\ \hline \forall c : Concept;\ pf : Feature;\ s : \mathbb{P}\, Feature \bullet \\ \quad c \underline{Mandatory} (pf, s) \Leftrightarrow pf \notin s \\ \quad \wedge ((c, pf) \in holds \Rightarrow (\forall f : s \bullet (c, f) \in holds)) \\ \quad \wedge ((c, pf) \notin holds \Rightarrow (\forall f : s \bullet (c, f) \notin holds)) \end{array}$$

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 set s should be included into the description of the same concept instance; otherwise none.

3.2.2 Optional

Optional defines features that *may or may not* be included into the description of a concept instance, if their parent is included. It can be defined formally as follows.

$$
\begin{array}{l}
\mathit{Optional} : \mathit{Concept} \leftrightarrow (\mathit{Feature} \times \mathbb{P}\,\mathit{Feature}) \\
\hline
\forall\, c : \mathit{Concept};\; pf : \mathit{Feature};\; s : \mathbb{P}\,\mathit{Feature} \bullet \\
\quad c\ \underline{\mathit{Optional}}\ (pf, s) \Leftrightarrow pf \notin s \\
\quad \wedge\ ((c, pf) \notin \mathit{holds} \Rightarrow (\forall f : s \bullet (c, f) \notin \mathit{holds}))
\end{array}
$$

The above states that if the parent feature pf of a set of *Optional* features s is not included in a feature configuration, there should be no features included from the set s in the same concept instance; otherwise, the choice is free.

3.2.3 Alternative

Alternative defines that *exactly one* feature from a set of features can be included into the description of concept instance, if its parent is included; otherwise none. Its formal definition is as follows.

$$
\begin{array}{l}
\mathit{Alternative} : \mathit{Concept} \leftrightarrow (\mathit{Feature} \times \mathbb{P}\,\mathit{Feature}) \\
\hline
\forall\, c : \mathit{Concept};\; pf : \mathit{Feature};\; s : \mathbb{P}\,\mathit{Feature} \bullet \\
\quad c\ \underline{\mathit{Alternative}}\ (pf, s) \Leftrightarrow pf \notin s \\
\quad \wedge\ ((c, pf) \in \mathit{holds} \Rightarrow (\exists_1 f : s \bullet (c, f) \in \mathit{holds})) \\
\quad \wedge\ ((c, pf) \notin \mathit{holds} \Rightarrow (\forall f : s \bullet (c, f) \notin \mathit{holds}))
\end{array}
$$

3.2.4 Or

Or defines that *one or more* features from a set of features can be included into the description of a concept instance, if their parent is included; otherwise none. Its formal definition is as follows.

$$
\begin{array}{l}
\mathit{Or} : \mathit{Concept} \leftrightarrow (\mathit{Feature} \times \mathbb{P}\,\mathit{Feature}) \\
\hline
\forall\, c : \mathit{Concept};\; pf : \mathit{Feature};\; s : \mathbb{P}\,\mathit{Feature} \bullet \\
\quad c\ \underline{\mathit{Or}}\ (pf, s) \Leftrightarrow pf \notin s \\
\quad \wedge\ ((c, pf) \in \mathit{holds} \Rightarrow (\exists f : s \bullet (c, f) \in \mathit{holds})) \\
\quad \wedge\ ((c, pf) \notin \mathit{holds} \Rightarrow (\forall f : s \bullet (c, f) \notin \mathit{holds}))
\end{array}
$$

3.2.5 Normalized feature types

As mentioned in Section 2.2, there are two other types of features in a feature diagram, i.e., optional-alternative and optional-or features. An optional-alternative feature type denotes that one or more features in a set of alternative-features is optional. From a concept instance point of view, it has the same result as all the features in the alternative set are optional. An optional-or feature type denotes that one or more features in a set of or-features is optional. This is the same as all the feature in the or-feature set are optional, which can be further simplified as each feature in the or-feature set is optional individually. Thus the optional-or features is a redundant feature type, which can be replaced by a set of individual optional features. The above is called normalization on feature diagrams [1]. Therefore, given any feature model, it can be represented by its normalized form that only contains five possible different type of features, i.e., *Mandatory*, *Optional*, *Alternative*, *Or* and *OptionalAlternative*. We define the *OptionalAlternative* feature type as follows.

$$
\begin{array}{l}
\mathit{OptionalAlternative} : \mathit{Concept} \leftrightarrow (\mathit{Feature} \times \mathbb{P}\,\mathit{Feature}) \\
\hline
\forall\, c : \mathit{Concept};\; pf : \mathit{Feature};\; s : \mathbb{P}\,\mathit{Feature} \bullet \\
\quad c\ \underline{\mathit{OptionalAlternative}}\ (pf, s) \Leftrightarrow pf \notin s \\
\quad \wedge\ ((c, pf) \in \mathit{holds} \Rightarrow ((\exists_1 f : s \bullet (c, f) \in \mathit{holds}) \\
\qquad \vee\ (\forall f : s \bullet (c, f) \notin \mathit{holds}))) \\
\quad \wedge\ ((c, pf) \notin \mathit{holds} \Rightarrow (\forall f : s \bullet (c, f) \notin \mathit{holds}))
\end{array}
$$

The above defines that if the parent feature pf of its optional-alternative child set s is held by a concept instance, *at most one* feature from the set s can be included in the same concept instance; otherwise none.

3.3 Additional constraints among features

So far, we have provided formal semantics for each component in a feature diagram. However, a feature model not only consists of the relationships presented in a feature diagram, but also includes additional constraints among the features that indicates valid combinations in a feature model. In other words, some features may be dependent on the presence of other features in a concept instance. We identified two relations, i.e., *Requires* and *Excludes*, to capture the additional constraints among features as follows.

3.3.1 Requires

Requires defines a relationship that the selection of a feature in the description of a concept instance *requires* the selection of other features. Its formal definition is as follows.

$$
\begin{array}{l}
\mathit{Requires} : \mathit{Concept} \leftrightarrow (\mathit{Feature} \times \mathbb{P}\,\mathit{Feature}) \\
\hline
\forall\, c : \mathit{Concept};\; f_1 : \mathit{Feature};\; s : \mathbb{P}\,\mathit{Feature} \bullet \\
\quad c\ \underline{\mathit{Requires}}\ (f_1, s) \Leftrightarrow \\
\quad (c, f_1) \in \mathit{holds} \Rightarrow (\forall f_2 : s \bullet (c, f_2) \in \mathit{holds})
\end{array}
$$

The above states that if a feature f_1 is selected by a concept instance, then all features in the set s must also be selected by the same instance.

3.3.2 Excludes

Excludes defines a relationship that the selection of a feature in a concept instance *excludes* the selection of the others. It can be defined as follows.

$$\begin{array}{l} Excludes : Concept \leftrightarrow (Feature \times \mathbb{P}\, Feature) \\ \hline \forall c : Concept;\ f_1 : Feature;\ s : \mathbb{P}\, Feature \bullet \\ \quad c\ \underline{Excludes}\ (f_1, s) \Leftrightarrow \\ \quad (c, f_1) \in holds \Rightarrow (\forall f_2 : s \bullet (c, f_2) \notin holds) \end{array}$$

In summary, a feature diagram captures the hierarchical relationships between a parent feature and its direct child features. And these relationships can be well described by the five different types of features, i.e., *Mandatory*, *Optional*, *Alternative*, *Or* and *Optional-Alternative*. The two relations, *Requires* and *Excludes*, capture the additional constraints among any features in the feature model. Such additional constraints can be dependencies, mutual exclusions and so on, which can be expressed by using the combination of the above two relations.

4 Meta-level theorems & proofs in Z/EVES

Since the formal semantics presented in the previous Section will serve as a precise and rigorous formal foundation for the graphical feature modeling language, we need to ensure that it is logically sound and it captures the meaning intended. Therefore, 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 enough to reasoning about feature model properties using Z/EVES efficiently and effectively. As suggested in [10], it is necessary to provide "a sufficient stock of theorems" to describe properties and facts of definitions and to express the relationship of several definitions so their proof 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` and the library of theorems is put into 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 theorem *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 $(c,f) \in holds$ can be rewritten as $f \in holds(\!|\ \{c\}\ |\!)$, the relational image of *holds* applied to $\{c\}$.

Z Section *feature_theory*, **parents:** *feature*

theorem disabled grule ConceptIsFeature
$\forall c : Concept \bullet c \in Feature$

theorem rule imageHoldsRule
$\forall c : Concept;\ f : Feature \bullet$
$\quad (c,f) \in holds \Leftrightarrow f \in holds(\!|\ \{c\}\ |\!)$
...

end of Z Section *feature_theory*

The library was 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 as follows.

From the semantics defined in the previous Section, 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, *altTranRule*3 states that if the parent f of a set of *Alternative* features s is held by a concept c, then the intersection of the 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.

theorem altTranRule3
$\forall c : Concept;\ f, g : Feature;\ s : \mathbb{P}\, Feature \bullet$
$(c,f) \in holds \wedge (c, (f, s)) \in Alternative \wedge$
$g \in s \wedge (c, g) \in holds \Rightarrow$
$\quad \{g\} = s \cap holds(\!|\ \{c\}\ |\!)$

This theorem is too distant from the definition of *Alternative* to be proved directly. Hence, to prove it, we developed a few auxiliary theorems, two of which are shown here.

theorem altTranRule1
$\forall c : Concept;\ f : Feature;\ s : \mathbb{P}_1\, Feature \bullet$
$(c,f) \in holds \wedge (c, (f, s)) \in Alternative \Rightarrow$
$\quad (\exists g : s \bullet s \cap holds(\!|\ \{c\}\ |\!) = \{g\})$

theorem altTranRule2
$\forall c : Concept;\ f : Feature;\ s : \mathbb{P}_1\, Feature \bullet$
$(c,f) \in holds \wedge (c, (f, s)) \in Alternative \Rightarrow$
$\quad \#(s \cap holds(\!|\ \{c\}\ |\!)) = 1$

proof[*altTranRule*2]
simplify;
*use altTranRule*1$[c := c, f := f, s := s]$;
prove;
equality substitute $s \cap (holds\ (\!|\ \{c\}\ |\!))$;
use sizeUnit[*Feature*]$[x := g]$;
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 *altTranRule*1 to conclude that $s \cap holds(\| \{c\} \|)$ is equivalent with $\{g\}$, whose cardinality is 1.

In the library, besides theorems about particular definitions in the semantics, there are also theorems about the relationship between various definitions. 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 $holds(\| \{c\} \|)$, the features held by c.

theorem manAltRule3
$\forall c : Concept;\ f : Feature;\ s_1, s_2 : \mathbb{P}_1\ Feature \mid$
$(c,f) \in holds \wedge (c,(f,s_1)) \in Mandatory \wedge$
$s_1 \cap s_2 \neq \emptyset \wedge (c,(f,s_2)) \in Alternative\ \bullet$
$(\exists g : Feature \bullet \{g\} = s_1 \cap s_2)$

The library contains 40 theorems, all of which are proved by Z/EVES[1]. These 40 theorems serve two purposes:

- They help to prove the correctness of the formal semantics by stating and proving desirable properties about definitions in the semantics. Proving the theorems give us more confidence that the definitions in section *feature* capture the intended meaning.
- They can be used to verify feature models in the future. As stated earlier, in Z/EVES, simple facts should be proved with relative ease and automation. These theorems can help Z/EVES to assume facts and rewrite some predicates to a closer form towards the goal. Future feature models in Z will themselves be put into sections with *feature_theory* as parent or ancestor.

Although Z/EVES itself can be used to verify feature model, however, as a theorem prover, the verification process requires much user interaction and expertise. A more automated solution is desired. In the next Section, we demonstrate the use of the Alloy Analyzer for automated feature model verification.

5 Feature model verification in Alloy

The semantics presented in the previous Sections provide a formal basis for the automated verification of feature models, an important task in domain engineering. We propose to use Alloy Analyzer to perform verification for three reasons: Firstly, Alloy is a light-weight formal modeling language based on first-order logic. It can be viewed as a subset of Z. We can easily encode the first-order semantics of the feature language in it. Secondly, Alloy Analyzer is a fully automated reasoning tool that requires no user interactions. Thirdly, it provides simulation and checking functionalities that are good for finding counterexamples and identifying source of the inconsistencies in the model. In this Section, we use the feature model for the Key Word in Context (KWIC) index systems to illustrate the verification process.

5.1 The Key Word in Context feature model

The KWIC (Key Word in Context) problem was used by Parnas [8] to contrast different criteria for modular software decomposition. Since its introduction, the problem has been used in several studies to illustrate the benefits of different software architectural styles. We use it as a case study to demonstrate the effectiveness of our approach in verifying feature models using Alloy. Parnas formulated the KWIC problem as follows:

"*The KWIC [Key Word in Context] index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be 'circularly shifted' by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a list of all circular shifts of all lines in alphabetical order.*"

All the KWIC systems are similar in the sense that they satisfy the basic requirements as stated above. However, there are also differences across KWIC systems related to functional requirements, design decisions and implementation details. We have identified the following features of KWIC systems:

- Input/Output - Original lines can be read from a text file, or from system console, or both. Sorted lines can be output to a text file, or to system console, or both.
- Circular Shift - Original lines are 'circularly shifted' by repeatedly removing the first word and appending it at the end of the line.
- Shift Processing - Line shifting can be performed on each line as it is read from the input device or after all the lines have been read.
- Shift Data - Circular shifts can be stored explicitly (as a set of strings) or implicitly (as pairs of index and offset).

[1]A full list of the semantics and theorem library can be found at: http://www.comp.nus.edu.sg/~liyf/feature_semantics/

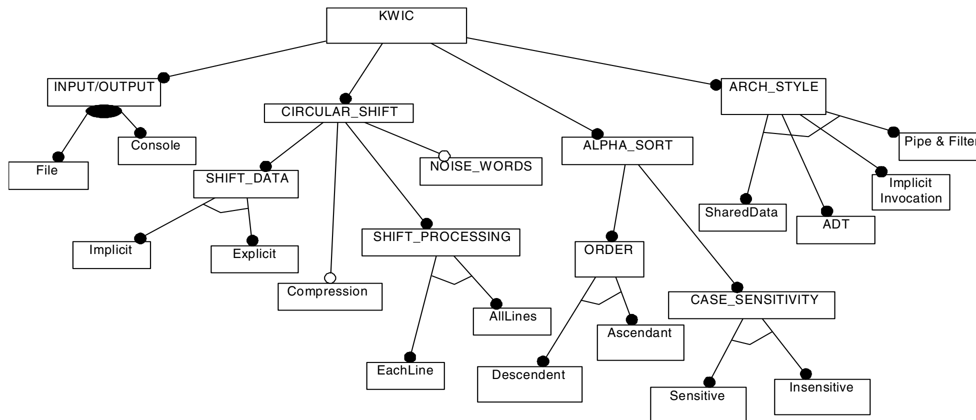

Figure 2. A feature model for the Key Word in Context index systems.

- Noise Words - In some KWIC systems, circular shifts that start with noise words (such as a, the, and, etc.) should be eliminated.

- Compression - The circular shifts could be stored in compressed or uncompressed form.

- Alphabet Sort - All circular shifts of all lines are sorted in alphabetical order.

- Order - The alphabetically sorted lines could be in descendent or ascendant order.

- Case Sensitivity - Case sensitivity may or may not be taken into account during sorting lines.

- Architecture Style - As described by Shaw and Garlan [11], the architecture style for a KWIC system could be Shared Data (functional decomposition), ADT (Abstract Data Types), Implicit Invocation, and Pipe & Filters.

Based on the above feature classification, a feature diagram for the KWIC systems can be defined as shown in Figure 2. We also know from our knowledge of the KWIC systems that not all the combinations of the features described in the above feature diagram (Figure 2) are valid in a KWIC implementation. Some additional constraints among the features are described as follows.

- Compression and ShiftData - We compress the circularly shifted lines when ShiftData is explicit (when circular shifts are stored as a set of strings).

- ShiftProcessing and ArchStyle - The Pipe & Filter architectural style is limited to the sequential/batch processing, precluding the use of the incremental processing. Thus the decision on ShiftProcessing depends on the decision on ArchStyle.

- ShiftData and ArchStyle - If the architectural style is Pipe & Filter, each filter (such as Input, Circular Shift and Alphabet Sort) has to keep a copy of lines, thus the circular shifts can be only stored explicitly. So the decision on ShiftData is also dependent on ArchStyle.

5.2 Encoding and presenting feature models in Alloy

As we mentioned earlier, Alloy is a light-weight formal modeling language based on first-order logic. We can easily encode the first-order semantics of the feature language in Alloy and perform automated feature model verification using the Alloy Analyzer.

```
module feature/FeatureModel
sig Feature {}
disj sig Concept extends Feature {
 holds : set Feature
}{ this in holds }

fun Mandatory(c:Concept,pf:Feature,s:set Feature)
{pf !in s
 pf in c.holds => all f:s | f in c.holds
 pf ! in c.holds => all f:s | f !in c.holds
}
...
```

Note that we encode the semantic of feature language in the Alloy module '*FeatureModel.als*' as above. Users can reuse this module to construct their Alloy-based feature models for specific domains (concepts). For example, the *KWIC* feature model in Figure 2 can be presented in Alloy as follows.

```
open feature/FeatureModel
disj sig KWIC extends Concept{}
static disj sig CircularShift extends Feature{}
static disj sig ShiftData extends Feature{}
...
fact {
 Mandatory(KWIC, KWIC, InputOutput+
```

```
  CircularShift+AlphaSort+ArchStyle) }
fact {
 Alternative(KWIC, ArchStyle, SharedData+
        ADT+ImplicitInvocation+PipeFilter) }
 ...
fact {Requires(KWIC, Compression, Explicit)}
fact {Requires(KWIC, PipeFilter, AllLines) }
 ...
}
```

Note the *open* command refers to the semantic definitions of feature diagram in the '*FeatureModel.als*' module. In Alloy, a *fact* is a set of predicates that constrains the values of the sets and relations. In the above example, the first group of *fact* statements specify different feature types in the feature model; while the second group of *fact* statements specify the additional constraints among features in the *KWIC* feature model defined in the previous subsection.

After transforming a feature diagram into its corresponding Alloy-based formal model, we can use the Alloy Analyzer to check various *KWIC* feature configurations readily.

5.3 Verifying the KWIC concept instances

We perform automated feature model verification through the analysis of the Alloy feature model. With the aid of Alloy Analyzer, we can verify whether a set of features selected from a feature model represents a valid instance. For invalid feature combinations, we can identify the source of unsatisfiability in the original model to point out where the conflicts are. For a given feature model, we can show whether such a model is solvable and list valid feature configurations in the model. We use the KWIC example to illustrate the various verification tasks that can be performed through Alloy as follows.

5.3.1 Checking a valid concept instance

To check whether a concept instance (configuration) is valid under a feature model in Alloy, we first use the negation technique to negate the statement as: there is no such instance existed under the feature model. If by running the Alloy Analyzer a counterexample can be found against this statement, we can say that the instance is a valid concept configuration. For example, if we want to check whether the concept instance listed below is valid, we make an assertion stating that this configuration does not exist, as follows.

```
assert Correctness1 {
 no c:KWIC | c.holds = c+InputOutput+File+
  Console+CircularShift+ShiftData+Implicit+
  NoiseWords+ShiftProcessing+EachLine+Order
  +AlphaSort+Descendent+CaseSensitivity
  +Sensitive+ArchStyle+SharedData
}
check Correctness1 for 25
```

Note that '25' is the total number of features in the *KWIC* model. The Alloy Analyzer checks whether this assertion holds by trying to find a counterexample. Alloy outputs '*Solution found*' and displays the instance, which means that Alloy has found a *KWIC* instance that has the above configuration. The result of the checking is shown in Figure 3. Thus we can conclude that our assertion does not hold and the above configuration is indeed valid.

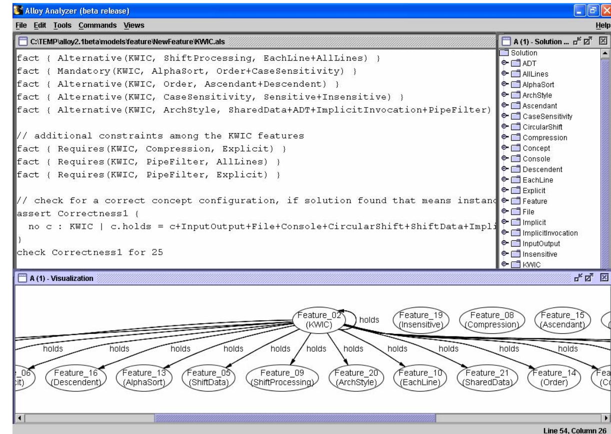

Figure 3. Checking a valid configuration.

5.3.2 Checking an invalid concept instance

We use the similar method to check an invalid configuration, such as listed below. We define an assertion as follows.

```
assert Correctness2{
 all c:KWIC | c.holds != c+InputOutput+
  File+CircularShift+ShiftData+Implicit+
  Compression+ShiftProcessing+AllLines+
  AlphaSort+Ascendant+CaseSensitivity+
  Insensitive+ArchStyle+Order+PipeFilter
}
check Correctness2 for 25
```

This time, the Alloy Analyzer could not find a solution. It outputs '*No solution found*', which means no counterexample could be found against the assertion statement. Thus we can conclude that our assertion holds and such a configuration for the *KWIC* feature model is not valid.

Besides discovering the existence of an invalid instance in a feature model, tracing where the inconsistency arises is also crucial for a reasoning tool to be practical. Without any tool support, identifying the cause of the inconsistency in a feature model could be frustrating. One possible systematic technique for finding the causes of inconsistent definitions is to manually remove individual knowledge information until any inconsistency is identified. This task can be lengthy and error prone. In the recent version of Alloy Analyzer, the 'unsatisfied core' functionality of the SAT solvers was utilized. It supports the *core extraction*, a new analysis

technique that helps to discover over-constraint in declarative models. This functionality can provide us some assistance for tracing the cause of an invalid concept instances and inconsistencies in a feature model. For example, if we use the '*determine unsat core*' function on the above invalid instance, the output is as follows.

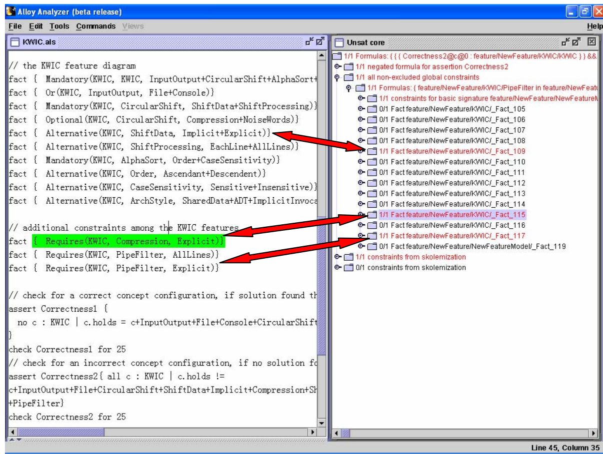

Figure 4. Checking the source of unsatisfiability.

Figure 4 shows how the Alloy Analyzer determines which facts caused the problem. In the right panel, clauses highlighted with red color are related to the conflicts in the model. Arrows were added in the figure to show the correspondence between the clauses in the right panel and the concepts in the left panel. After examining all red clauses, we found that the three clauses with arrows attached (in Figure 4) actually caused the inconsistency. Hence, the lack of solution was indeed due to the inconsistency between the assertions (facts) and their original definitions. In this case, the inconsistency is mainly caused by the contradiction of *Compression* requiring Explicit, *PipeFilter* architecture requiring *Explicit* and alternative features *Implicit* and *Explicit* in the *KWIC* model, as shown in Figure 4.

5.3.3 Checking solvability of a feature model

It is hard to assume that we could always select a valid configuration from a feature model. Furthermore, we cannot assume that the feature models that we wrote are always consistent by design. However, Alloy can verify the validity of the feature models by generating valid concept instances one at a time, as follows:

```
fun findInstance() {some KWIC}
run findInstance for 25
```

The *findInstance* predicate instructs the Alloy Analyzer to find some valid configurations in the *KWIC* model. By selecting a large enough scope (e.g., 25 in this example), we are able to validate the solvability of a feature model design. As for our *KWIC* model, a valid configuration is returned as the model is solvable. Alloy finds one solution at a time. Other possible solutions (valid configurations) can be explored through the 'next' function in Alloy. Solutions can be viewed visually through the visualization function provided by the Alloy Analyzer. In addition, if a feature model is inconsistent by design, we can use the 'unsatisfied core' functionality to determine where the source of inconsistency is, as illustrated in the previous subsection.

5.4 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.

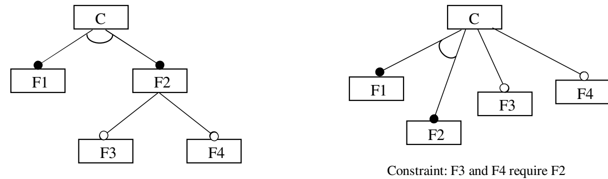

Figure 5. Equivalent feature models.

Considering the two models shown in Figure 5, we can see that these two feature models share the same feature configurations (C, F1), (C, F2), (C, F2, F3), (C, F2, F4), and (C, F2, F3, F4). Thus they are semantically equivalent. Note that the feature model on the right has an additional constraint as '*F3 and F4 require F2*'. To prove the equivalence, we formalize these two feature models in Alloy and make an assertion as follows:

```
disj sig C extends Concept{}
static disj sig F1 extends Feature{}
...
assert Equivalent{
 (Alternative(C, C, F1+F2) &&
  Optional(C, F2, F3+F4)) <=>
 (Alternative(C, C, F1+F2) &&
  Optional(C, C, F3+F4)&&
  Requires(C, F3, F2) &&
  Requires(C, F4, F2))
}
check Equivalent for 5
```

We state in the assertion *Equivalent* that the two feature models are semantically equivalent. Alloy cannot find a counter example against the statement. Thus, it verifies that our assertion is true, that the above two feature models are indeed semantically equivalent at the conceptual level.

6 Conclusions

In this paper, we presented an approach to formalizing and verifying feature models using formal reasoning techniques. The contributions of the paper lie in the following three main categories. Firstly, we provided a formal semantics for the feature model language using the first-order logic in Z. We specified formal definitions for the five feature types and two additional relations for feature modeling.

Secondly, we validated the correctness of the defined semantics using the Z/EVES theorem prover. A library of proof rules (theorems) are developed for supporting the meta-level reasoning for the feature modeling language. Z/EVES is an interactive theorem prover that supports the Z language, in which is the formal semantics built. By stating properties about the semantics formally in first-order logic and proving them in Z/EVES, we can ensure that it captures the intended meaning. Thus the defined feature semantics provide a precise and rigorous formal basis for the feature modeling process. It is based on such a formal semantics that automated formal reasoning and verification of feature models can be made possible.

Thirdly, we demonstrated that by encoding the semantics in Alloy, Alloy Analyzer can be used to verify the consistency of a given feature model, with the ability of generating counterexamples in the case of inconsistencies. We chose Alloy Analyzer to perform feature model verification for three reasons: firstly, Alloy is based on first-order logic, which makes the encoding straightforward; secondly, the verification process is fully automated, which frees the users from the tedious, error-prone manual reasoning tasks; thirdly, in case of an inconsistency in the model, Alloy Analyzer can generate counterexamples and 'unsatisfied core', which helps to identify the source of the inconsistency. A case study of the Key Word in Context (KWIC) index systems feature model was presented to illustrate the verifications process. We showed that the valid and invalid configurations of the KWIC feature model can be verified, the solvability of a feature model can be checked and the source of inconsistencies in a feature model can be identified. Furthermore, we also demonstrated that two different feature models can be checked for semantic equivalence at a conceptual level. In summary, our approach provided a variety of automated feature model verification facilities.

In the future, we plan to develop an integrated feature modeling environment that supports the construction of graphical feature models, the exchange of feature models using XML, and the transformation from the XML format into their corresponding Alloy feature models for automated verification.

References

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

[2] M. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *The Fifth International Conference on Software Reuse*, pages 76–85, Vancouver, BC, Canada, 1998.

[3] D. Jackson. Micromodels of software: Lightweight modelling and analysis with Alloy. Website, 2002. `http://alloy.mit.edu/reference-manual.pdf`.

[4] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy Constraint Analyzer. In *22nd International Conference on Software Engineering*, pages 730–733, Limerick, Ireland, 2000. ACM Press.

[5] K. C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[6] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[7] K. C. Kang, J. Lee, and P. Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 9:58–65, 2002.

[8] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[9] M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer-Verlag, 1997.

[10] M. Saaltink. The Z/EVES 2.0 User's Guide. Technical Report TR-99-5493-06a, ORA Canada, One Nicholas Street, Suite 1208 - Ottawa, Ontario K1N 7B7 - CANADA, Oct. 1999.

[11] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

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

[13] C. Turner, A. Fuggetta, L. Lavazza, and A. Wolf. A Conceptual Basis for Feature Engineering. *Journal of Systems and Software*, 49:3–15, 1999.

[14] D. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[15] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice-Hall, 1996.