Using First Order Logic to Validate Feature Model

Abdelrahman O. Elfaki, Somnuk Phon-Amnuaisuk, Chin Kuan Ho

Center of Artificial Intelligent and Intelligent computing,
Multimedia University, Cyberjaya, Malaysia

abdelrahman.osman.06@mmu.edu.my, somnuk.amnuaisuk@mmu.edu.my, ckho@mmu.edu.my

Abstract

Feature Model (FM) is approved as a successful technique to model variability in Software Product Line (SPL), therefore it is very important to produce error-free FM. Inconsistency in FM is addressed as key challenge in validation of FM. This paper completes the knowledge-base(KB) method for validating FM by defining a new operation, namely inconsistency-prevention. First the inconsistency in FM is categorized into four groups as a prerequisite process for inconsistency-prevention. Then the scalability of KB method for validating FM is tested. Empirical results for each operation are presented and discussed. The empirical results are employed to illustrate scalability and applicability of the proposed knowledge-base method.

1. Introduction and Motivations

FM is considering as one of the successful methods for applying variability in SPL [1]. Therefore it is very important to produce error-free FM, this process is non-feasible manually. Inconsistency detection is introduced in [2] as a research challenge. Inconsistency occurs from contradictions in constraint dependency rules. It is very complicated because it has different formats and it can occur between groups of features or between individual features. In our previous work [3], we introduced a method to validate a FM based on mapping FM to first order logic, one-to-one mapping and represent domain engineering as a KB. The proposed method [3] defines and provides auto-support for three operations for validating FM, namely: explanation, dead feature detection, and inconsistency-detection. The proposed operation inconsistency-detection can detect only inconsistency between individual features. In this paper we enhance this operation by illustrating how to map all inconsistency formats to one-to-one format. And rather than detect inconsistency, we improve the proposed method by defining a new operation that aims at preventing inconsistency in FM, this process is explore the KB(domain engineering) and according to specific states new rules added to KB. The overall contribution of the proposed method is the validating of FM within domain engineering process.

Scalability is one of the main factors that define the applicability of the methods that deal with FM. Empirical results approved (in literature) to test scalability. In this paper we test the scalability of the proposed method (four operations).

2. Related Work

Mannion [4] was the first to connect propositional formulas to feature models., but the model did not detect the dead features or inconsistency. Zhang et al.[5] defined a meta-model of FM using UML core package. Zhang only validated consistency-check during configuration. Benavides et al. [6] proposed a method for mapping feature model into constraint solver. This model was mainly developed for the analysis of SPL-based FM, rather than validating it. Batory[7] proposed a coherent connection between FMs, grammar and propositional formulas, represented basic FMs using context–free grammars plus propositional logic. This connection allows arbitrary propositional constraints to be defined among features and enables off-the-shelf satisfiable solvers to debug FM. Batory’s method validated FM within application engineering process (in product derivation), and it detected one type of inconsistency (one-to-one) and...
did not detect the dead features. Although Janota [8] used higher-order logic to reason feature models, unfortunately no real implementation has been described. Thang [9] defined a formal model in which each feature was separately encapsulated by a state transition model. The aim of the study is to improve consistency verification among features, there is no mention for inconsistency or dead features. Czarnecki [10] proposed a general template-based approach for mapping feature models, and used object-constraint language (OCL) in [11] to validated constraint rules. In [3] we proposed rules for consistency constraint check. These rules are different from other methods (to validate consistency check) by considering and dealing with variation point constraints.

Trinidad [12] defined a method to detect dead features based on finding all products and search for unused features. The idea is to automate error detection based on theory of diagnosis [13]. This model mapped FM to diagnose-model and used CSP to analyze FM. Our proposed method to detect dead features [3] has less cost than Trinidad’s because it searches only in three predefined cases, i.e. in domain engineering process. Validating FM in domain engineering is one of the main contributions of our proposed method. Segura et al. [14] used atomic set as a solution for the simplification of FMs. Segura scaled the work using random data generated by FAMA [15]. White et al. [16] proposed a method for debugging feature model configurations. This method scales as models with over 5,000 features are randomly generated by FAMA. Our proposed method is a collection of predicates; therefore it has high degree of flexibility, e.g. it can be partitioned regarding specific features. As we proved in [3], the proposed method is the first that deals with inconsistency. Moreover it addresses the validation operations (inconsistency-detection, dead feature detection, and explanation. In this study, we define inconsistency-prevention as a fourth operation applied to domain engineering (rather than configure a solution and validate it), which enhances the maturity of SPL. In the next section we illustrated the fourth operation (of the proposed KB method [3]) inconsistency prevention.


In our previous work [3], we defined and illustrated three operations: i) explanation, ii) dead feature detection, and iii) inconsistency-detection. In this section inconsistency-prevention is defined as fourth operation, and later experiments (which are designed to evaluate scalability of our proposed operations) are explained and results are discussed. In addition to validating existing FMs, the proposed method can be used to prevent inconsistency in FM by adding new relations (exclude/require).

The following parts of this section are defining the prerequisite process and illustrating the rules of inconsistency-prevention operations.

3.1. Prerequisite Process

The prerequisite process for inconsistency-prevention and inconsistency-detection operations is converting all forms of inconsistency into one-to-one relation form.

Inconsistency Forms

Inconsistency in FM, could be categorized in four groups:

Many-to-Many inconsistency:

In many-to-many inconsistency a set requires other set while the required set excludes the first one. E.g. 

\((\{A_1, A_2, \ldots, A_n\} \text{ requires } \{B_1, B_2, \ldots, B_m\}) \text{ and} \ (\{B_1, B_2, \ldots, B_m\} \text{ excludes } \{A_1, A_2, \ldots, A_n\})\)

Other possible scenario, a set can requires other set while some features of the required set excludes some features of the first one. e.g.:

\(((A, B, C) \text{ requires } (D, E, F)) \text{ and } ((G, F, H) \text{ excludes } (A, B, C)).\)

The constraint dependency could be between two or more sets.

Many-to-One inconsistency:

A set of features has constraint dependency relation (require/exclude) with one feature while this feature has a contradiction relation to this set or to some of its elements. e.g.

\(((A, B, C) \text{ requires } D) \text{ and } (D \text{ excludes } (B, C)).\)

One-to-Many inconsistency:

One feature has constraint dependency relation (require/exclude) with a set of features while this set has a contradiction relation to this feature.

One-to-One inconsistency:

One feature has a constraint dependency with one feature while the second feature has a contradiction relation to the first feature. e.g.\((A \text{ requires } B) \text{ and } (B \text{ excludes } A).\)

In [3], we defined and illustrated five rules to detect One-to-One inconsistency. To detect other forms of inconsistency we need first to extend Many-to-Many, Many-to-One, and One-to-Many to represented as feature-to-feature relation. The following rule extends forms of inconsistency to feature-to-feature relation:

\[ \forall_f A_i | 1 <= i <= n \text{ relation } B_j | 1 <= j <= m \implies A_i \text{ relation } B_j \]

Where relation represents constraint rule (requires/excludes).

Example:

Many-to-Many inconsistency \(((A, B) \text{ require } (D, E)) \text{ and } ((G, E) \text{ excludes } (A, B))\) can be extended to \(((A \text{ requires } D) \text{ and } (A \text{ excludes } A)) \text{ and } (B \text{ requires } E) \text{ and } (B \text{ requires } D)\) and \((G \text{ excludes } B) \text{ and } (E \text{ excludes } A)\) and (}
E excludes B)) feature-to-feature relation.

3.2. Inconsistency-prevention Rules

Inconsistency in feature model is a relationship between features that cannot be true at the same time[2]. To avoid inconsistency (prevent inconsistency) the following rules are proposed:

i. \( \forall x,y,z: \text{type}(x, \text{variant}) \rightarrow \text{type}(y, \text{variant}) \rightarrow \text{requires}_v(y,x) \rightarrow \text{requires}_v(y,z) \rightarrow \text{requires}_v(x,z) \)

ii. \( \forall x,y,z: \text{type}(x, \text{variationpoint}) \rightarrow \text{type}(y, \text{variationpoint}) \rightarrow \text{requires}_v(x,y) \rightarrow \text{requires}_v(x,z) \rightarrow \text{requires}_v(y,z) \)

iii. \( \forall x,y,z: \text{type}(x, \text{variant}) \rightarrow \text{requires}_v(x,y) \rightarrow \text{requires}_v(x,z) \rightarrow \text{requires}_v(y,z) \rightarrow \text{requires}_v(y,x) \)

iv. \( \forall x,y,z: \text{type}(x, \text{variant}) \rightarrow \text{requires}_v(y,x) \rightarrow \text{requires}_v(y,z) \rightarrow \text{requires}_v(x,z) \rightarrow \text{requires}_v(x,y) \)

v. \( \forall x,y,z: \text{type}(x, \text{variant}) \rightarrow \text{requires}_v(y,x) \rightarrow \text{excludes}_v(y,z) \rightarrow \text{excludes}_v(x,z) \)

vi. \( \forall x,y,z: \text{type}(x, \text{variant}) \rightarrow \text{requires}_v(y,x) \rightarrow \text{requires}_v(y,z) \rightarrow \text{excludes}_v(x,z) \rightarrow \text{excludes}_v(y,z) \)

vii. \( \forall x,y,z: \text{type}(x, \text{variant}) \rightarrow \text{requires}_v(y,x) \rightarrow \text{excludes}_v(y,z) \rightarrow \text{excludes}_v(x,z) \rightarrow \text{excludes}_v(y,z) \)

The outputs of this operation are new constraint dependency rules added to the KB (domain engineering) to sustain the consistency.

4. The Experiment

We developed an algorithm to generate random FMs (predicates form). We have three assumptions: i) each variation point and variant has unique name, ii) each variation point is orthogonal, and iii) all variation points have the same number of variants. The main parameters are the number of variants and the number of variation points. The remaining eight parameters (common variants, common variation points, variant requires variant, variant excludes variant, variation point requires variation point, variation point excludes variation points, variant requires variation point, and variant excludes variation point) are defined as a percentage. The number of variant-related parameters (such as common variant) is defined as a percentage of the number of variants. The number of variation point-related parameters (such as variant requires variation point) is defined as a percentage of the number of variation points. For each number of variant/variation point we made ten experiments, and calculated execution time as average. The experiments were done with the range (1000-20000) variants, and percentage range of 10%, 25%, and 50%.

4.1. Empirical Results

4.1.1. Explanation: To evaluate the scalability of this operation, we define additional parameter, the predicate select(\( V \)): where \( V \) is random variant. This predicate simulates user selection. Number of select predicate (defined as a percentage of number of variants) is added to the KB (domain engineering) for each experiment, and the variant \( V \) is defined randomly (within scope of variants). Figure 2 illustrates the average execution time.

4.1.2. Dead Feature Detection: Figure 3 illustrates the average execution time. For (20,000) variants and 50% of constraint dependency rules, the execution time is 3.423 minutes which can be considered good time. White et al.[16] scaled their work by 5,000 feature in one minute. The output of each experiment is a result file containing the dead variants.

4.1.3. Inconsistency-Detection: Figure 4 illustrates the average execution time to detect inconsistency in FM range from 1000 to 20,000 variants.

4.1.4 Inconsistency-prevention: new dependency rules (requires/excludes) should be added to the KB to prevent inconsistency. Figure 5 illustrates the average execution time to prevent inconsistency in FM range from 1000 to 20,000 variants.

5. Conclusion

The proposed method deals with the complexity of validating product line based feature model. It is the first method that detects an inconsistency in FM of all types. Moreover, it explores the existing relations and prevents future inconsistency. The validation process (dead feature detection, inconsistency-detection, and inconsistency-prevention) should be applied to domain engineering which guarantees error free domain engineering. Error-free domain engineering (one of the main contributions of the proposed method) promises generation of valid applications. Many methods are applying empirical results to test scalability by generating random FMs [5, 13, 16]. Comparing with literature, our test range (1000 – 20,000 features) is
sufficient to test scalability.

Figure 3: Dead-feature Detection Result

Figure 4: Inconsistency-Detection

Figure 5: Inconsistency-prevention Results

6. References