A Formal Approach to Verify Mapping Relation in a Software Product Line

Tonny Kurniadi Satyananda, Danhyung Lee, and Sungwon Kang

Information and Communications University
119 Munji-ro Yuseong-gu, Daejeon 305-732, Korea
{tonny, danlee, kangsw}@icu.ac.kr

Abstract

In software product line development, consistency among artifacts is important because commonalities and variabilities increase the complexity of relations among artifacts. For small scale models, the relations among elements can be easily identified and tracked by manually analyzing the descriptions of models. But when the complexity of models is high, a more systematic approach is required for identifying traceability information and verifying consistency between models. In this paper, by utilizing Formal Concept Analysis (FCA) and Prototype Verification System (PVS), we present a formal approach for identifying traceability and verifying consistency between feature model and component and connector view of software architecture.

1. Introduction

A software product line is “a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [1]. Traceability among software product line development artifacts is more complex than traceability among single software development artifacts, because it deals with commonalities and variabilities. Therefore, consistency verification among artifacts becomes more important. Usually, traceability among artifacts is not just simple one-to-one mapping relation, but it also includes one-to-many mapping relation, many-to-one mapping relation, or even many-to-many mapping relation. These two conditions contribute to complex traceability and later lead to complex verification. When the complexity of traceability is high, a formal approach will be useful to help verifying the consistency. People tend to avoid dealing with many-to-many mapping relation, because of its complexity. But it is not easy to completely exclude many-to-many relation in artifacts mapping.

The main goal of our work is to establish a framework for automated formal identification of traceability and automated formal verification of consistency between feature model and architecture model. By identifying the mapping between feature model’s elements and architecture model’s elements, we can identify the traceability between the two models. Later, the identified mapping is used to verify consistency between feature model and architecture model.

In this research, to reduce checking complexity, we focus only on feature model and architecture model in core asset development. Product derivation verification such as feature configuration or specific product architecture conformance to product line architecture is not considered. We use component and connector view to represent software architecture model. We utilize Formal Concept Analysis (FCA) to identify traceability and we utilize Prototype Verification System (PVS) to verify models consistency.

The rest of this paper is organized as follows. Section 2 presents background concepts. Section 3 then describes our approach to identify traceability and to verify consistency between feature model and architecture model. Section 4 discusses a case study of digital watch product line and mapping relations. In Section 5, we conclude this paper and present future direction.

2. Background

2.1. Feature Model

Feature modeling is a method for describing commonalities and variabilities in software product line. It was introduced for the first time by Kang et al. in 1990 in Feature Oriented Domain Analysis (FODA) [2]. Since then, many researches have been done to suggest improvement over feature model. Some
researchers have proposed a formal textual language to describe feature model [3] [4].

Kang et al. [2] define features as the attributes of a system that directly affect end-users. A feature model consists of a feature diagram and other information such as rationale, constraint, and dependency rule. A feature diagram is a tree-like notation that shows the hierarchical structure of features. The root of the tree is referred as a concept node. There are several definitions of feature variability: mandatory feature, optional feature, and alternative feature. Feature dependency (static constraint) is classified into ‘requires’ or ‘mutex’ (excludes).

2.2. Architecture Model

According to Bass et al. [5], software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Component and Connector (C&C) views define models consisting of elements that have some runtime presence (such as processes, objects, clients, servers, and data stores) and their interactions [6]. Variability of element in this view can be classified as mandatory, optional, or variant. Architecture Description Language (ADL) is used to describe architecture model in textual format. A large number of ADLs have been proposed, e.g. ACME [7]. Asikainen et al. in [8] present a comparison analysis of ADLs for software product lines (ACME, Wright, and Koala).

2.3. Formal Concept Analysis

Formal Concept Analysis (FCA) is a theory of data analysis which identifies conceptual structures among data sets. It uses lattice theory to provide a way to group and discuss objects based upon their common attributes. FCA has been applied in many part of software development, e.g. for architectural element matching [9].

In FCA, the basic object of manipulation is the formal context (C) which is defined as the ordered triple (O, A, R) where O is the extent (or set of objects), A is the intent (or set of attributes), and R is a relation between the extent and intent \( R \subseteq O \times A \). From the relation R, a lattice is computed which allows examination of concepts shared among objects. A concept can be thought of as a maximal set of shared attributes. Further details on concept lattices theory can be found in [10].

2.4. Prototype Verification System (PVS)

PVS is a prototype verification system for development and analysis of formal specifications [11]. The PVS system consists of a specification language, a parser, a type-checker, a prover, specification libraries, and various browsing tools. A PVS specification consists of a collection of parameterized theories. Theories are the basic units of modularity, used to package the various elements of the specification, including its types, axioms, constants, and theorems. The PVS language is based on strongly typed higher-order logic. It includes a rich collection of basic types, including booleans, integers, strings, enumerated types, records, tuples, functions, and sets. Paper [12] provides references for formalization of software architecture in PVS.

3. The Approach

To identify traceability and to verify consistency between feature model and architecture model, we proposed the following steps as shown in Fig. 1:

1) Extract functional decomposition.

To identify traceability, we need a basis for matching the elements. We use functional decomposition of a feature and functional decomposition of an architecture element to find matching elements. We define functional decomposition as a list of functionality descriptions which describes the functional capability. The description should be provided in feature description and architecture description in a consistent way.

Figure 1. Traceability Identification and Consistency Verification Approach
Different keywords are interpreted as different functional capability. And, to simplify the extraction process, each model description should conform to a specific definition language. We use modified Forfamel [4] (by adding functional decomposition declaration) to describe feature model, and ACME to describe architecture model.

2) Build concept lattice.

To build concept lattice, we use Christian Lindig’s concept package [13]. This concept package implements mathematical algorithm to generate concept lattice graph and it will give a result in simple textual format.

3) Analyze concept lattice to identify traceability.

Concept lattice must be traversed to identify functional decomposition-based mapping between feature elements and architecture elements based on some mapping criteria. When we trace mapping relations starting from each feature element, we can identify mapping-pair of feature element and architecture element. Mapping rules consist of general rules between FCA concepts which are derived to identify mapping-pairs. Depending on how we implement graph traversal algorithm to find mapping relations, we can further identify one-to-one mapping, one-to-many mapping, many-to-one mapping, and many-to-many mapping relations.

4) Build PVS specification for every model description.

Feature model description and architecture model description should be translated into PVS specification. These specifications are basis for formal verification. They represent ‘what-is’ described in the two models.

5) Build PVS mapping specification based on identified traceability.

Identified mapping relation between feature model and architecture model should also be modeled in PVS specification. This specification is also a basis for formal verification. It represents ‘what-is’ described in the two models. Mapping relation is declared as a set of feature and architecture-element pairs.

6) Check variability and dependency consistencies using PVS.

Based on model specifications and identified mapping, develop theorems to verify variability and dependency consistencies. These theorems represent ‘what-should-be’ consistent between the two models.

We define consistency theorems based on “a feature is mapped to architecture element(s)” viewpoint to cover one-to-one mapping, one-to-many mapping relations, and many-to-one mapping relations between feature and architecture elements. To cover consistency theorems for many-to-many mapping relation, many-to-many mapping relation is decomposed into one-to-many mapping and many-to-one mapping relations. This means theorems for one-to-many mapping relation and theorems for many-to-one mapping relation are used for many-to-many mapping relation. Next section shows the proof of many-to-many mapping relation decomposition.

4. Case Study

In order to illustrate how our approach can be applied, we use a portion of Digital Watch product line example shown in Fig. 2.

Figure 2. Portion of Digital Watch Feature Model and Architecture Model
Without loss of generality, and also to reduce the complexity of definition and decomposition proof, mapping relation between feature model and architecture model is defined as feature-component mapping.

From feature description and architecture description, we extract functional decompositions and build concept lattice. By applying mapping rules to the lattice, we get feature to architecture component mapping. The identified mapping is shown in Fig. 3.

Next, we have to build PVS specifications. For simplicity, we describe feature model, architecture model, mapping specification, and consistency theorems in one PVS theory specification. Due to lack of space, we only show a portion of type specifications here in Fig. 4. Then, to verify mapping relation, we build variability and dependency consistency theorems. The following is an example of excludes-dependency checking theorem:

- If feature f1 excludes feature f2, then for each component c1 which implements feature f1 and for each component c2 which implements feature f2, c1 excludes c2:

\[
\text{EXC_REQ_TH: THEOREM} \\
\text{FORALL (f1:(dw_fd`feats) | } f_a(f1)`excludes /= emptyset): \\
\text{FORALL (f2:(f_a(f1)`excludes)):} \\
\text{FORALL (c1:(dw_ad`components) | member((f1,c1),dw_map)),} \\
\text{c2:(dw_ad`components) | member((f2,c2),dw_map)):} \\
\text{member(c2,c_a(c1)`excludes)}
\]

This theorem is shown to be true with (grind).

Many-to-many mapping relation has to be handled differently. To reduce complexity, many-to-many mapping relation should be decomposed into a combination of one-to-many mapping and many-to-one mapping relations.

Many-to-many mapping relation exists when there are two different features which are mapped to one component, and also there is a feature that maps to two different components. In other words, a feature requires more than one component, and a certain component is used to implement two or more features. Fig. 5 illustrates two basic cases of many-to-many mapping relation, which are two-to-two mapping relations. Those two cases are similar.

Using binary relation notation, we define feature set, component set, and mapping relation between feature and component as follow:

- \( F \) is a feature set,
- \( C \) is a component set, and
- \( M \) is a heterogeneous binary relation between \( F \) & \( C \), a subset of Cartesian Product between \( F \) and \( C \):

\[
M : P(F \times C)
\]

Specific sets of mapping relation are defined as follow: (note that, ‘==’ notation is used to define abbreviation [14], and ‘:’ notation is used to define member set)

---

**Figure 3. Identified Mapping**

**Figure 4. Portion of PVS Type Specification of Digital Watch**
Figure 5. Simple Many-To-Many Mapping Relations

- The set of all one-to-one mapping relations in M:
  \( \text{Rel}_{1-1}[M] \equiv \{(f,c): M \mid (\forall(f_a,c_a): M \bullet f_a = f \Leftrightarrow c_a = c)\} \)

- The set of all one-to-many mapping relations in M:
  \( \text{Rel}_{1-n}[M] \equiv \{(f,c): M \mid (\exists(f_1,c_1), (f_2,c_2), \ldots,(f_n,c_n): M \bullet c_1 \neq c \land c_2 \neq c \land \ldots \land c_n \neq c)\} \)

- The set of all many-to-one mapping relations in M:
  \( \text{Rel}_{m-1}[M] \equiv \{(f,c): M \mid (\exists(f_1,c_1), (f_2,c_2), \ldots, (f_m,c_m): M \bullet f_1 \neq f \land f_2 \neq f \land \ldots \land f_m \neq f)\} \)

- The set of all many-to-many mapping relations in M: (m-n)
  \( \text{Rel}_{m-n}[M] \equiv \{(f,c): M \mid (\exists(f_1,c_1), (f_2,c_2), \ldots, (f_m,c_m): M \bullet (f_1 \neq f_2 \neq \ldots \neq f_m) \land c_1 \neq c_2 \neq \ldots \neq c_m \land (f,c) = (f_1,c_1) \lor (f,c) = (f_2,c_2) \lor \ldots \lor (f,c) = (f_m,c_m)) \}\)

Note that, both examples are covered in above many-to-many mapping relation definition. The right-hand example can be obtained by exchanging \( f_1 \) with \( f_2 \) and \( c_1 \) with \( c_2 \) in left-hand example. Furthermore, the simplified (2-2) definition can be used to represent (m-n) definition, because (m-n) mapping relation can be build by expanding (2-2) mapping relation or combining several (2-2) mapping relations. This (2-2) definition will be used in decomposition proof.

The following theorems show decomposition of many-to-many mapping relations.

Theorem 1:
For heterogeneous binary relation M which has many-to-many mapping relation,
\( \text{Rel}_{m-n}[M] \subset \text{Rel}_{1-n}[M] \cap \text{Rel}_{m-1}[M] \)

Proof:
Given any \((f,c) \in \text{Rel}_{m-n}[M]\), by definition, there are \((f_1,c_1), (f_1,c_2), \text{ and } (f_2,c_2) \in \text{Rel}_{m-n}[M]\). Then, there are three possibility of \((f,c)\):

- \((f,c) = (f_1,c_1)\). Because there is \((f_1,c_2)\) and \(c_1 \neq c_2\), then \((f,c) \in \text{Rel}_{1-n}[M]\).
- \((f,c) = (f_1,c_2)\). Because there is \((f_1,c_1)\) and \(c_1 \neq c_2\), then \((f,c) \in \text{Rel}_{1-n}[M]\), and because there is \((f_2,c_2)\) and \(f_1 \neq f_2\), then \((f,c) \in \text{Rel}_{m-1}[M]\).
- \((f,c) = (f_2,c_2)\). Because there is \((f_1,c_2)\) and \(f_1 \neq f_2\), then \((f,c) \in \text{Rel}_{m-1}[M]\).

So \( \text{Rel}_{m-n}[M] \subset \text{Rel}_{1-n}[M] \cap \text{Rel}_{m-1}[M] \)

Theorem 2:
For heterogeneous binary relation M which has many-to-many mapping relation,
\( \text{Rel}_{m-n}[M] = \{(f,c) : \text{Rel}_{1-n}[M] \cup \text{Rel}_{m-1}[M] \}\)

Proof:
Given any heterogeneous binary relation M which has many-to-many mapping relation, \( \text{Rel}_{m-n}[M] \) has at least three elements, let’s say, \((f_1,c_1), (f_1,c_2), \text{ and } (f_2,c_2) \in \text{Rel}_{m-n}[M]\). Then, they should be members of \( \text{Rel}_{l_a}[M]\) and \( \text{Rel}_{l_b}[M]\):

- \((f_1,c_1) \in \text{Rel}_{l_a}[M] \land \text{Rel}_{l_b}[M] = \{f_1\}\)
- \((f_1,c_2) \in \text{Rel}_{l_a}[M] \land \text{Rel}_{l_b}[M] = \{f_2\}\)
- \((f_2,c_2) \in \text{Rel}_{l_a}[M] \land \text{Rel}_{l_b}[M] = \{f_1\}\)

Therefore, \( \text{Rel}_{m-n}[M] \) is the union of \( \text{Rel}_{l_a}[M] \) and \( \text{Rel}_{l_b}[M] \) which complies to the theorem.

For examples shown on Fig. 5, many-to-many relation can be defined simply as (2-2):
\( \text{Rel}_{2-2}[M] \equiv \{(f,c) : M \mid (\exists(f_1,c_1), (f_2,c_2), (f_3,c_3): M \bullet f_1 \neq f_2 \neq f_3 \neq f \land c_1 \neq c_2 \neq c_3 \land (f,c) = (f_1,c_1) \lor (f,c) = (f_2,c_2) \lor (f,c) = (f_3,c_3)) \}\)
So $\text{Rel}_{\pi}(M) = \\
\{ (f,c) : \text{Rel}_{\pi}(M)(\exists (f_1,c_1) : \text{Rel}_{\pi}(M) \cap \text{Rel}_{\pi}(M)[f = f_1] \cup (f,c) : \text{Rel}_{\pi}(M)[c = c_2]) \\
(\exists (f_2,c_2) : \text{Rel}_{\pi}(M) \cap \text{Rel}_{\pi}(M)[c = c_2]) \}

Based on these proofs, PVS consistency theorems for one-to-many mapping relation and for many-to-one mapping relation can be used for many-to-many mapping relation.

The following are examples of variability consistency theorems for digital watch product line:

- If feature $f$ is a mandatory feature, then each component which implements feature $f$ must be a mandatory component:
  \begin{verbatim}
  VAR_MAN_TH01 : THEOREM
  FORALL (f:(dw_fd`feats) | f /= dw_fd`concept):
    f_a(f)`var_of = mandatory =>
    FORALL (c:(dw_ad`components) | member((f,c),dw_map)):
      c_a(c)`var_of = mandatory
  \end{verbatim}
This theorem is shown to be true with (grind).

- If there is a mandatory feature in a set of features which is mapped to a component, then the component must be a mandatory component:
  \begin{verbatim}
  VAR_MAN_TH02 : THEOREM
  FORALL (f1,f2:(dw_fd`feats)) ,
    (c:(dw_ad`components)) ;
    (f1 /= f2) AND member((f1,c),dw_map)
    AND ((f_a(f1)`var_of = mandatory)
    OR (f_a(f2)`var_of = mandatory)) =>
    c_a(c)`var_of = mandatory
  \end{verbatim}
This theorem is shown to be true with (grind).

5. Conclusion

In this paper, we have presented a formal approach to identifying traceability and verifying consistency between feature model and architecture model using Formal Concept Analysis and Prototype Verification System. In our approach, many-to-many mapping relation is covered as well as one-to-one, one-to-many, or many-to-one mapping relations. With this approach, we can automate traceability identification and consistency verification between the two models.

For future work, we plan to perform PVS specification translation work, consistency theorems building, and consistency verification automatically. Currently, the translation process is performed manually and the consistency verification is performed semi-manually through a sequence of proof commands. We will also explore other important properties and investigate the applicability of this approach on more complex systems.

6. Acknowledgment

This work was supported by the Industrial Technology Development Program funded by the Ministry of Commerce, Industry and Energy (MOCIE, Korea).

7. References