Formal Verification of Consistency between Feature Model and Software Architecture in 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

During software development process, software artifacts are produced. Consistency among these artifacts should be verified to ensure error-free product. In software product line development, consistency becomes more important because commonalities and variabilities increase the complexity of relationship among artifacts. In this paper, we present a formal approach to verification of consistency between feature model and component and connector view of software architecture. By utilizing Prototype Verification System (PVS), we introduce our model of feature description and architecture description, and illustrate the consistency verification approach using a digital watch product line example.

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. But, complex traceability leads to complex verification. When the complexity of traceability is high, a formal approach will be useful to help verifying the consistency.

The main goal of our work is to establish a framework for automated formal verification of consistency between feature model and architecture model. By modeling feature description, architecture description, and mapping relationship between feature model and architecture model in PVS, we establish a basis for formal verification. These specifications represent ‘what-is’ described in the two models. To verify consistency, we develop variability consistency theorems and dependency theorems as ‘what-should-be’ consistent between the two models. Then, based on those specifications, we mechanically check the variability consistency and dependency consistency between the two models.

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. For simplicity, mapping relationship between the models is defined as feature-component mapping.

The rest of this paper is organized as follows. Section 2 presents basic concepts and our verification approach. In Section 3, we illustrate our approach by defining model specifications of digital watch product line. Section 4 discusses mechanical verification of the specification by constructing formal theorems and how they are proved. Section 5 discusses related work. In Section 6, we conclude this paper and present the future direction of our research.

2. Key Concepts and Model Specifications

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.

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 to as the concept node.

In FODA, there are several definitions of feature variability: mandatory features must be selected whenever their parent is selected; an optional feature may or may not be selected when its parent is selected; an alternative feature consists of a set of alternatives of which exactly one must be selected whenever the parent feature is selected. Feature dependency (static constraint) is classified into ‘requires’ or ‘mutex’ (excludes). In our approach, we define feature model specification based on FODA. Fig. 1 shows an example of Digital Watch product line feature model.

2.2. Architecture Model

According to Bass et al. [3], 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 [4]. 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 [5]. Askainen et al. in [6] present a comparison analysis of ADLs for software product lines (ACME, Wright, and Koala). Fig. 2 shows an example of Digital Watch product line architecture model (high level).

2.3. Prototype Verification System (PVS)

PVS is a prototype verification system for development and analysis of formal specifications [7]. 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 [8] provides references for formalization of software architecture in PVS.

2.4. Model Specifications

In order to formally verify consistency between feature model and architecture model, we need their descriptions in a suitable formal specification, e.g. PVS specification.

For simplicity, we describe feature model, architecture model, feature-to-architecture mapping, and consistency theorems in one PVS theory specification. We begin by defining basic type definitions used in our model.

---

**Figure 1. An example of Digital Watch Product Line Feature Model**

**Figure 2. An example of Digital Watch Product Line Architecture Model**
The following declaration defines the name of the elements in an enumeration set. Connector types, connector roles, and variability types are defined similarly.

```plaintext
fname_type  : TYPE = {<list-of-feature-name>}
cpname_type : TYPE = {<list-of-component-name>}
cnname_type : TYPE = {<list-of-connector-name>}
pname_type  : TYPE = {<list-of-port-name>}
conn_kind   : TYPE = {<list-of-connector-type>}
role        : TYPE = {<list-of-connector-role>}
var_type    : TYPE = {mandatory, optional, alternative, variant}
```

We use a tree representation with linked table (recursive node representation) to describe feature model. Feature element is defined as a type represented by natural numbers, which mapped to a specific feature attribute. Feature attribute consists of feature name, feature variability, a link to parent feature, a set of child features, a set of required-feature dependency, and a set of excluded-feature dependency.

```plaintext
feature : TYPE+ = nat
fattr   : TYPE = [# name     : fname_type,
var_of   : var_type,
parent   : feature,
children : setof[feature],
requires : setof[fattr],
excludes : setof[feature] #]
```

Instance of null feature and instance of null feature attribute are also defined in the theory. Due to lack of space, we do not include them here. Next, feature diagram is declared as a type consisting of concept feature, a set of features, and a set of feature attributes. To increase readability in consistency theorems, feature and feature attribute relationship is defined in a simple table declaration similar to feature and feature attribute relationship table.

```plaintext
pcp_tuple      : TYPE = [port,connector,port]
architecture_d : TYPE = [# components  : setof[component],
cattrs      : setof[cattr],
connectors  : setof[connector],
attachments : setof[pcp_tuple] #]
```

Mapping relationship among feature elements and architecture components is declared as a set of feature-component pairs.

```plaintext
fc_tuple : TYPE = [feature,component]
fcmap    : TYPE = setof[fc_tuple]
```

The declaration of variability consistency theorems and dependency consistency theorems is described in Section 4.

### 3. Case Study: Digital Watch

In order to illustrate how our approach can be applied, we use an example of Digital Watch product line. Because the models shown in Fig. 1 and Fig. 2 are too big, we will focus on consistency verification on a portion of Digital Watch models related to Alarm and Speaker functionalities.

Example of feature instances, feature attribute instances, feature to feature-attribute relationship table (f_a), and feature diagram instance are shown below.

```plaintext
feat01 : feature = 1
feat02 : feature = 2
% and other feature declarations
fattr01 : fattr =
```
Similar instances declaration approach applied to architecture element instances, component to component-attribute relationship table (c_a), and architecture diagram instance.

Feature to architecture component mapping (dw_map) is declared using member set add construct according to the mapping shown in Fig. 3. Note that concept feature is not mapped to any component.

4. Verification

After the model descriptions have been translated, we construct mathematical description of consistency verification in the form of theorems, and then, demonstrate that they follow from the descriptions. In this paper, we only focus on variability consistency and dependency consistency between feature model and architecture model. Other properties checking in only feature model or other properties checking in only architecture model are not considered.

When we use PVS prover, we have to guide the proof checker with a series of commands corresponding to the deductive rules of higher-order logic. In our case, many of the theorems can be proved using (grind). This command repeatedly performs rewrites, definition expansions, skolemizations, automatic quantifier instantiations, and propositional simplifications.

We check consistency by defining theorems based on ‘features are mapped to architecture components’ viewpoint. The theorems are defined for one-to-one, one-to-many, and many-to-one mappings. Many-to-many mapping checking can be built by analyzing the mapping set in more detail. In this paper, we show only theorems defined for one-to-one and one-to-many mappings. Consistency theorems for one-to-many mapping cover consistency theorems for one-to-one mapping.

For each mapping specification, we define two kinds of checking theorems, strong checking and weak checking. Strong checking is used to verify strong mapping consistency, which means the checking

Figure 3. Feature to Architecture Map
properties of all mapped elements should be the same. Weak checking does not enforce similarity of checking properties for all mapped elements in one-to-many mapping. The following are examples of variability consistency checking.

**Theorem 1a** – strong mandatory property checking: if feature \( f \) is a mandatory feature, then all components which implement feature \( f \) must be mandatory components:

\[
\text{VAR\_MAN\_TH\_S \ : \ THEOREM}
\]
\[
\forall (f : (\text{dw\_fd\`feats}) | f \neq \text{dw\_fd\`concept}):
\quad \forall (c : (\text{dw\_ad\`components}) | \exists \text{member((f,c),dw_map)}):
\quad c \_a(c) \_\text{var\_of} = \text{mandatory}
\]

This theorem is shown to be true with \((\text{grind})\).

**Theorem 1b** – weak mandatory property checking: if feature \( f \) is a mandatory feature, then from all components which implement feature \( f \), there is at least one mandatory component:

\[
\text{VAR\_MAN\_TH\_W \ : \ THEOREM}
\]
\[
\forall (f : (\text{dw\_fd\`feats}) | f \neq \text{dw\_fd\`concept}):
\quad \exists (c : (\text{dw\_ad\`components}) | \exists \text{member((f,c),dw_map)}):
\quad c \_a(c) \_\text{var\_of} = \text{mandatory}
\]

This theorem is shown to be true with \((\text{grind} : \text{if-match nil}) (\text{grind} : \text{if-match all}) (\text{grind} : \text{if-match all})\).

**Theorem 2a** – strong optional property checking: if feature \( f \) is an optional feature, then all components which implement feature \( f \) must be optional components:

\[
\text{VAR\_OPT\_TH\_S \ : \ THEOREM}
\]
\[
\forall (f : (\text{dw\_fd\`feats}) | f \neq \text{dw\_fd\`concept}):
\quad \forall (c : (\text{dw\_ad\`components}) | \exists \text{member((f,c),dw_map)}):
\quad c \_a(c) \_\text{var\_of} = \text{optional}
\]

This theorem cannot be proved because our example does not conform to this strong theorem.

**Theorem 2b** – weak optional property checking: if feature \( f \) is an optional feature, then from all component which implements feature \( f \), there is at least one optional component:

\[
\text{VAR\_OPT\_TH\_W \ : \ THEOREM}
\]
\[
\forall (f : (\text{dw\_fd\`feats}) | f \neq \text{dw\_fd\`concept}):
\quad \exists (c : (\text{dw\_ad\`components}) | \exists \text{member((f,c),dw_map)}):
\quad c \_a(c) \_\text{var\_of} = \text{optional}
\]

This theorem is shown to be true with \((\text{grind} : \text{if-match nil}) (\text{grind} : \text{if-match all}) (\text{grind} : \text{if-match all}) (\text{grind} : \text{if-match all})\). Due to lack of space, those theorems are not shown here. The following are examples of dependency consistency checking.

**Theorem 4a** – strong requires-dependency checking: if feature \( f_1 \) requires feature \( f_2 \), then for each component \( c_1 \) which implements feature \( f_1 \) and for each component \( c_2 \) which implements feature \( f_2 \), \( c_1 \) requires \( c_2 \):

\[
\text{DEP\_REQ\_TH\_S \ : \ THEOREM}
\]
\[
\forall (f_1 : (\text{dw\_fd\`feats}) | f_1 \neq \text{dw\_fd\`concept}):
\quad \exists (f_2 : (\text{f\_a(f1)\`requires}) | \exists \text{member((f_1,c_1),dw_map)},
\quad \exists (f_2 : (\text{dw\_ad\`components}) | \exists \text{member((f_2,c_2),dw_map))}:
\quad \text{member}(c_2,c \_a(c_1) \_\text{requires})
\]

This theorem cannot be proved because our example does not conform to this strong theorem.

**Theorem 4b** – weak requires-dependency checking: if feature \( f_1 \) requires feature \( f_2 \), then there is component \( c_1 \) which implements feature \( f_1 \) and there is component \( c_2 \) which implements feature \( f_2 \), and \( c_1 \) requires \( c_2 \):

\[
\text{DEP\_REQ\_TH\_W \ : \ THEOREM}
\]
\[
\forall (f_1 : (\text{dw\_fd\`feats}) | f_1 \neq \text{dw\_fd\`concept}):
\quad \exists (f_2 : (\text{f\_a(f1)\`requires}) | \exists \text{member((f_1,c_1),dw_map)},
\quad \exists (f_2 : (\text{dw\_ad\`components}) | \exists \text{member((f_2,c_2),dw_map))}:
\quad \text{member}(c_2,c \_a(c_1) \_\text{requires})
\]

This theorem is shown to be true with \((\text{grind} : \text{if-match nil}) (\text{grind} : \text{if-match best}) (\text{instantiate} + ("comp35" "comp43")) \text{ (grind)}\).

**Theorem 5a** – strong excludes-dependency checking: if feature \( f_1 \) excludes feature \( f_2 \), then for each component \( c_1 \) which implements feature \( f_1 \) and for each component \( c_2 \) which implements feature \( f_2 \), \( c_1 \) excludes \( c_2 \):

\[
\text{EXC\_REQ\_TH\_S \ : \ THEOREM}
\]
\[
\forall (f_1 : (\text{dw\_fd\`feats}) | f_1 \neq \text{dw\_fd\`concept}):
\quad \exists (f_2 : (\text{f\_a(f1)\`excludes}) | \exists \text{member((f_1,c_1),dw_map)},
\quad \exists (f_2 : (\text{dw\_ad\`components}) | \exists \text{member((f_2,c_2),dw_map))}:
\quad \text{member}(c_2,c \_a(c_1) \_\text{excludes})
\]

This theorem is shown to be true with \((\text{grind})\).

**Theorem 5b** – weak excludes-dependency checking: if feature \( f_1 \) excludes feature \( f_2 \), then there is component \( c_1 \) which implements feature \( f_1 \) and there is component \( c_2 \) which implements feature \( f_2 \), and \( c_1 \) excludes \( c_2 \):

\[
\text{EXC\_REQ\_TH\_W \ : \ THEOREM}
\]
\[
\forall (f_1 : (\text{dw\_fd\`feats}) | f_1 \neq \text{dw\_fd\`concept}):
\quad \exists (f_2 : (\text{f\_a(f1)\`excludes}) | \exists \text{member((f_1,c_1),dw_map)},
\quad \exists (f_2 : (\text{dw\_ad\`components}) | \exists \text{member((f_2,c_2),dw_map))}:
\quad \text{member}(c_2,c \_a(c_1) \_\text{excludes})
\]

This theorem is shown to be true with \((\text{grind})\).
f_a(f1) \text{excludes} \neq \text{emptyset}:
\forall (f2 : (f_a(f1) \text{excludes}):
\exists (c1 : (dw_ad'\text{components}) | 
\text{member}((f1, c1), dw_map),
(c2 : (dw_ad'\text{components}) | 
\text{member}((f2, c2), dw_map)):
\text{member}(c2, c_a(c1) \text{excludes})
\)

This theorem is shown to be true with (grind :if-match nil) (grind :if-match best) (grind :if-match best).

With these theorems, we rigorously check the consistency between feature model and architecture model. We consider the strong checking theorems sufficient to establish full consistency between the two models. The weak checking theorems can be used to establish partial consistency between the two models.

5. Related Work

There are some researches focusing on feature dependencies and their relationship in software product line architecture.

K. Lee and K. C. Kang [9] extended the feature modeling to analyze feature dependencies that are useful in the design of reusable and adaptable product line components, and present design guidelines based on the extended model.

C. Zhu et al. [10] analyzed domain requirements dependencies’ influence on product line architecture, and proposed formal mapping rules from requirements to feature and from feature to architecture. But it’s not clear whether their mapping rules can be applied for automated verification or not.

D. Dhungana [11] reported an ongoing research in designing and implementing product line variability models, where focus lies in treating features and architectural elements as parts of an integrated model. But, they haven’t devised any mechanism to verify consistency of the models automatically.

The primary contribution of this paper is to establish a basis for automated formal verification of consistency between feature model and architecture model, by modeling model specifications and then verifying their consistency using PVS.

6. Conclusion

In this paper, we presented a formal approach to verification of consistency between feature model and software architecture in software product line. Using the case study of Digital Watch, we showed example of model description in PVS specification and consistency checking theorems. Our formal proofs are conducted interactively with PVS theorem prover. Our approach can be used as core part of automated verification of consistency between feature model and architecture model.

For future work, we plan to perform the PVS specification translation work and consistency theorems building automatically. Currently, the translation process is performed manually, and it is limited to simple models. For large models, optimized and automated process is necessary. We also will explore other important properties and investigate the applicability of this approach on more complex systems.

7. Acknowledgment

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

8. References