

# A Use Case Based Approach to Feature Models' Construction

Bo Wang, Wei Zhang, Haiyan Zhao, Zhi Jin, Hong Mei

*Key Laboratory of High Confidence Software Technology, Ministry of Education of China  
Institute of Software, School of EECS, Peking University, Beijing, 100871, China  
{wangbo07, zhangw, zhhy, zhijin}@sei.pku.edu.cn, meih@pku.edu.cn*

## Abstract

*In the research of software reuse, feature models have been widely adopted to organize the requirements of a set of applications in a software domain. However, there still lacks an effective approach to minimizing analysts' participation in feature models' construction. In this paper, we propose a use case based semi-automatic approach to the construction of feature models. The basic idea of this approach is to first construct a set of feature models for individual applications (called application feature models, AFMs) in a software domain, then adjust, and merge the set of AFMs to form a feature model for this domain (called a domain feature model, DFM). The main characteristic of this approach is that it provides a set of rules and algorithms to make the construction of AFMs (from use cases) and the construction of DFMs (by merging a set of AFMs) be carried out automatically. A running example is used to illustrate the main characteristic and the feasibility of this approach.*

## 1. Introduction

In the research of software reuse, feature models have been widely adopted to capture, organize and reuse the requirements of a set of similar applications in a software domain. To fully take advantage of feature models, the first thing is to construct them, which usually involves systematic commonality and variability analysis to the focused software domains.

Several methods have been proposed to facilitate feature models' construction [3,4,5]. For example, in FODA [4] and FORM [5], a set of guiding principles are provided to help analysts identify and organize features in a software domain. In FeatuRSEB [3], a process of constructing feature models is proposed, in which, a domain use case model is first created by analyzing a set of similar applications in a domain, and

then the feature model is derived from the domain use case model.

However, there still lacks an effective approach to minimizing analysts' participation in feature models' construction. Since that the construction of a feature model usually involves quantitative analysis of a set of applications in a software domain, the quality of the constructed feature model heavily depends on analysts' personal knowledge and experience on the domain. Therefore, it will be valuable if there is an approach that could minimize analysts' participation in feature models' construction by automating key activities in feature models' construction.

In this paper, we propose a use case based semi-automatic approach to feature models' construction, with the purpose of automating the activities of feature-oriented commonality and variability modeling. This approach consists of three main steps: first, construct a set of feature models for individual applications (called application feature models, AFMs) from use cases of these applications, with the support of a set of discovery rules; second, adjust the AFMs with the help of a set of adjusting rules and conflicts checking rules; third, merge the set of adjusted AFMs to form a feature model for the domain (called a domain feature model, DFM). The main characteristic of this approach is that it provides a set of rules and algorithms to make the construction of AFMs (from use cases) and the construction of DFMs (by merging a set of AFMs) be carried out automatically. A running example from the web store domain is used to illustrate the main characteristic and the feasibility of this approach.

The remainder of this paper is organized as follows. Section 2 gives some preliminary knowledge and introduces a running example. Section 3 devotes to descant our approach step by step. Section 4 illustrates our approach with the web store system case study. Related work is discussed in Section 5. Finally, Section 6 concludes this paper with a short summary and future work.

## 2. Preliminaries

In this section, we introduce two metamodels for use cases and feature models, respectively. The two metamodels serve as the bases of our approach.

### 2.1. A Metamodel of Use Cases

Use cases are widely adopted to capture the requirements of software applications from the users' perspective. To derive features from them, we propose a structural description method for use cases. Figure 1 shows the metamodel of the use cases used in our approach.

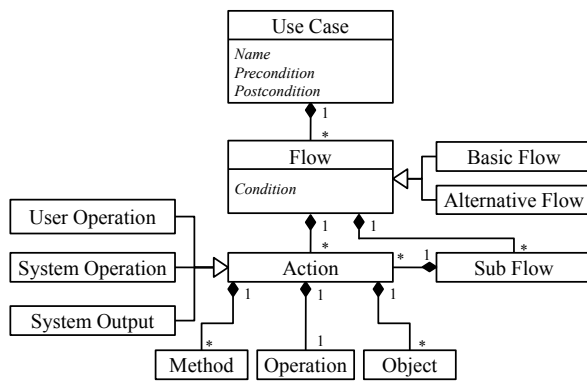


Figure 1. The metamodel of use cases

According to the metamodel, a use case, with attributes of *name*, *precondition*, and *postcondition*, is composed of a set of flows. Flows are classified into two categories: *basic flows* and *alternative flows*. A *basic flow* describes the most common implementation of a use case [8], while an *alternative flow* describes the alternative implementation of a use case [8]. A flow may consist of several *sub flows*, which describe partial implementations of the flow.

In the metamodel, flows or sub flows are composed of *actions*. An *action* is what the actor or the system does in the flows. Actions are classified into three categories: *user operations*, *system operations* and *system outputs*. *User operations* describe what actors do in the interactive process. *System operations* describe what the system does in the background. *System outputs* describe the output of the system operations.

An action consists of three parts: *object*, *operation* and *method*. An *object* is any resource that can be operated in a use case. An *operation* is what can be done to the object. A *method* is the way to operate an object.

### 2.2. The Metamodel of Feature Models

Figure 2 shows the metamodel of the feature models based on our previous work [6,9]. In this metamodel, a feature model consists of a set of features and two kinds of relationships between features, namely refinements and constraints.

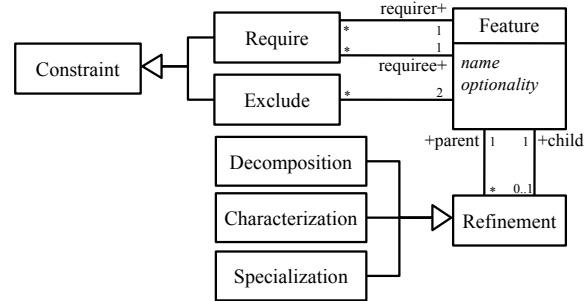


Figure 2. The metamodel of feature models

A feature is a software characteristic with sufficient user or customer value, which essentially denotes a cohesive set of individual requirements. The *optionality* attribute indicates whether a feature can be removed from the current feature model when its parent feature has been bound. This attribute has two values: *mandatory* and *optional*.

A *constraint* describes the dependencies between features. Two kinds of binary constraints are defined in [4], namely *requires* and *excludes*. If feature *A* *requires* feature *B*, it means that *B* cannot be removed from the current feature model when *A* is not removed. If feature *A* *excludes* feature *B*, it indicates that at most one of them can be bound in the same context.

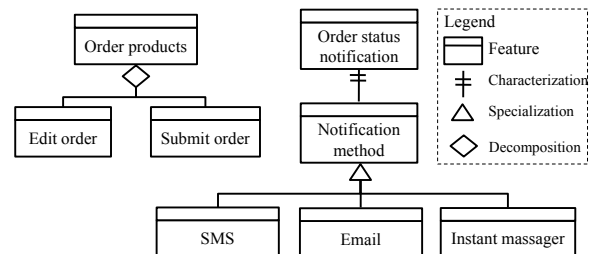


Figure 3. Examples of refinement relationships

There are three kinds of refinement relationships between features: *decomposition*, *characterization* and *specialization*. Refining a feature into its constituent features is called *decomposition* [4]. For example, the feature *Order products* in the web store domain can be decomposed into two child features: *Edit order* and

**Table 1. Use case “Place an order”**

Name: Place an order		
Postcondition: The order is submitted.		
User Operation	System Operation	System Output
Basic Flow		
1.(Open) <sub>operation</sub> the (cart) <sub>object</sub> .	2.(Check) <sub>operation</sub> the (login status) <sub>object</sub> .(A1)	3.(Display) <sub>operation</sub> all the (products) <sub>object</sub> .
4.(Update) <sub>operation</sub> the (quantity) <sub>object</sub> .(S1)	5.(Calculate) <sub>operation</sub> the (total price) <sub>object</sub> . 6.(Update) <sub>operation</sub> the (total price) <sub>object</sub> .	7.(Display) <sub>operation</sub> all the (products) <sub>object</sub> .
8.(Enter) <sub>operation</sub> the (claim code) <sub>object</sub> . 9.(Enter) <sub>operation</sub> the (security code) <sub>object</sub> . 10.(Redeem) <sub>operation</sub> the (gift certificate) <sub>object</sub> .	11.(Check) <sub>operation</sub> (validity of the gift certificate) <sub>object</sub> . (A2) 12.(Update) <sub>operation</sub> the (total price) <sub>object</sub> .	
13.(Enter) <sub>operation</sub> the (promo code) <sub>object</sub> . 14.(Apply) <sub>operation</sub> the (promo code) <sub>object</sub> .	15.(Check) <sub>operation</sub> the (validity of the promo code) <sub>object</sub> . 16.(Update) <sub>operation</sub> the (total price) <sub>object</sub> .(A3)	
17.(Enter) <sub>operation</sub> the (shipping address) <sub>object</sub> .(S2)	18.(Check) <sub>operation</sub> the (validity of the shipping address) <sub>object</sub> . (A4)	19.(Display) <sub>operation</sub> the (payment method selection window) <sub>object</sub> .
20.(Choose) <sub>operation</sub> (Credit Card) <sub>object</sub> , (PayPal) <sub>object</sub> , or (Mail Payment) <sub>object</sub> .		21.(Display) <sub>operation</sub> the (payment method) <sub>object</sub> . 22.(Display) <sub>operation</sub> the (shipping method selection window) <sub>object</sub> .
23.(Choose) <sub>operation</sub> (UPS) <sub>object</sub> or (FedEx) <sub>object</sub> .	24.(Calculate) <sub>operation</sub> the (shipping fee) <sub>object</sub> . 25.(Update) <sub>operation</sub> the (total price) <sub>object</sub> .	26.(Display) <sub>operation</sub> the (shipping method) <sub>object</sub> .
27.(Submit) <sub>operation</sub> the (order) <sub>object</sub> .		28.(Display) <sub>operation</sub> the (submit success message) <sub>object</sub> .
Alternative Flows (A1) Condition: The customer has not logged in.		
		1.(Display) <sub>operation</sub> the (login window) <sub>object</sub> .
Alternative Flows (A2) Condition: The gift certificate is invalid.		
		1.(Display) <sub>operation</sub> the (gift certificate error message) <sub>object</sub> .
Alternative Flows (A3) Condition: The promo code is invalid.		
		1.(Prompt) <sub>operation</sub> the (promo code error message) <sub>object</sub> .
Alternative Flows (A4) Condition: The shipping address is invalid.		
		1.(Display) <sub>operation</sub> the (shipping address error message) <sub>object</sub> (by popping up a dialogue box) <sub>method</sub> .
Sub Flows (S1)		
1.(Select) <sub>operation</sub> a (product) <sub>object</sub> . 2.(Enter) <sub>operation</sub> the (quantity) <sub>object</sub> .		
Sub Flows (S2)		
1.(Enter) <sub>operation</sub> the (First Name) <sub>object</sub> , (MI) <sub>object</sub> , (Last Name) <sub>object</sub> , (City) <sub>object</sub> , (Address) <sub>object</sub> , (Zip code) <sub>object</sub> , (State) <sub>object</sub> , and the (phone) <sub>object</sub> .		

*Submit order* (Figure 3). Refining a feature by identifying its attribute features is called *characterization* [9]. For example, the feature *Order status notification* in the web store domain can be characterized by attribute feature *Notification method* (Figure 3). Refining a feature into further detailed features is called *specialization* [4]. For example, the feature *Notification method* has three specialized features: *SMS*, *Email*, and *Instant Massager* (Figure 3). Through refinement, features with different abstract levels and granularities can form a hierarchy structure.

### 2.3. A Running Example

The running example used in this paper is based on web store systems. In a web store, the customers browse items, add the items they would like to order into the shipping cart and submit the order. It needs many use cases to describe the requirements of a web store system; however, due to the limitation of space,

we will take use case “Place an order” as a running example, as shown in Table 1. The use case is initiated by a request from the customer. Then the shipping and payment details are entered. After the legality checking of these details, the customer can submit the order.

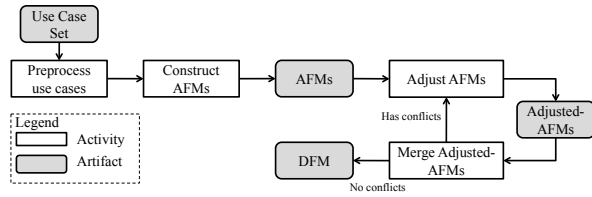
## 3. Use Case Based Construction of Features Models

In this section, we first give an overview of our approach, and then present the four steps of our approach in detail.

### 3.1. An Overview of the Process

Figure 4 exhibits the overall process of our approach. The input is a set of use cases, which describe similar functions of a set of domain specific applications. The output is a DFM that represents

partial characteristics of the domain. Most of the characteristics are described implicitly in the use cases. The whole construction process consists of four steps.



**Figure 4. The overall process of feature models' construction**

The first step is to identify similar operations, objects, and relationships between the objects by analyzing the set of the use cases (See section 3.2). This step serves as a preparation for the construction of AFMs.

The second step is to derive one AFM from each use case (See section 3.3). This step is carried out automatically based on eight discovery rules. For each use case, the derived AFM represents the corresponding characteristics of the application.

The third step is to eliminate the improper features and relationships by checking and adjusting the AFMs (See section 3.4). Three kinds of operations are provided to adjust AFMs; a set of heuristic rules are proposed to help adjust AFMs and identify conflicts between AFMs, respectively. With the help of these operations and rules, AFMs can be checked and adjusted semi-automatically.

The fourth step is to merge the Adjusted-AFMs into a DFM automatically (See section 3.5). For each set of use cases, the DFM represents the corresponding characteristics of the domain.

### 3.2 Preprocess use cases

In the input use case set, all the use cases are stated using the metamodel presented in Section 2.1. To derive AFMs from use cases, the use cases are preprocessed in two activities. In the first activity, two kinds of relationships between objects are identified for each use case. The two kinds of relationships are listed as follows:

- **'Is-a' relationship.** For any two objects  $A$  and  $B$ , there is a 'Is-a' relationship between  $A$  and  $B$  if  $B$  is a generalization of  $A$ , with  $B$  the parent and  $A$  the child. For instance, "payment method" (parent object) is a generalization of "mail payment" (child object).
- **'Has-a' relationship.** For any two objects  $A$  and  $B$ ,

there is a "Has-a" relationship between  $A$  and  $B$  if  $B$  is a part of  $A$ , with  $A$  the parent and  $B$  the child. For instance, "claim code" (child object) is a part of "gift certificate" (parent object).

For each use case, these relationships are used later to construct the corresponding AFM.

In the second activity, similar operations and similar objects in different use cases are assigned with identical names, which will be used when we adjust and merge the AFMs.

### 3.3. Construct AFMs

Having identified similar operations, similar objects and relationships between objects, one AFM can be automatically derived from each use case. We identify eight discovery rules to guide the construction of AFMs. These rules are classified into three categories and are listed as follows:

#### 1. Feature discovery rules:

- *F1: 'Discover features by indentifying operation and objects in an action.'*

The operation on each object in an action is regarded as a feature. The name of the feature is: "<operation> <object>".

For example, in use case "Place an order", "open" and "cart" are identified as operation and object of action  $I$  in basic flow, respectively. In this situation, there exists a feature named "Open cart", as shown in Figure 5(a).

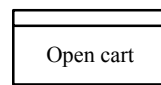
- *F2: 'Discover features by identifying the method of an action.'*

The method of an operation is regarded as a feature. The name of the feature is: "<method>".

For example, in use case "Place an order", "by popping up a dialogue box" is identified as the method of action  $I$  in alternative flow 4. The method is regarded as a feature, as shown in Figure 5(b).

#### Action $I$ in basic flow:

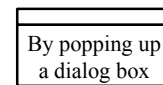
(Open)<sub>operation</sub> the (cart)<sub>object</sub>



(a)

#### Action $I$ in alternative flow 4:

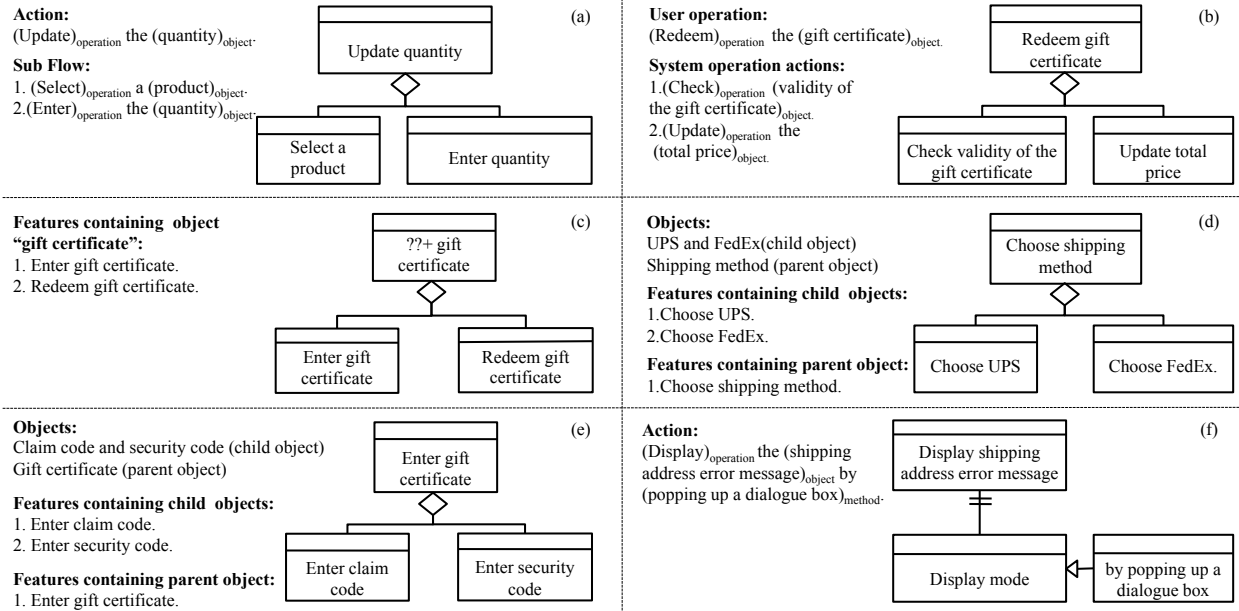
(Display)<sub>operation</sub> the (shipping address error message)<sub>object</sub> (by popping up a dialog box)<sub>method</sub>.



(b)

**Figure 5. Examples of feature discovery rules**

As described in rule  $F1$  and  $F2$ , a feature is named



**Figure 6. Examples of refinement relationship discovery rules**

by referencing the operation, object and method of an action. For those features that have not been discovered by applying rule *F1* and *F2*, they will be discovered later during the process of identifying the decomposition relationships between features.

## 2. Decomposition relationship discovery rules:

- *D1: 'Discover decomposition relationships by exploring sub flows.'*

In use cases, an action can be linked to a sub flow that describes this action in detail. The features generated from actions of the sub flow are the constituent features of the feature generated from the action that is linked to the sub flow.

For example, action “Update the quantity of the product” is linked to sub flow *I* in use case “Place an order”. Features generated from actions in sub flow *I* compose the feature generated from the action “Update the quantity of the product”, as shown in Figure 6(a).

- *D2: 'Discover decomposition relationships by exploring system operations.'*

Some user operations describe functions that the actor expects in the future system. System operations following the user operation may describe how the system achieves the expected function. The features generated from the system operations are the constituent features of the feature generated from the user operation.

For example, user operation “Redeem the gift certificate” is followed by two system operations.

Features generated from these two actions compose the feature generated from action “Redeem the gift certificate”, as shown in Figure 6(b).

- *D3: 'Discover decomposition relationships by identifying features whose names reference the same object.'*

For a set of features whose names reference the same object, a new feature named “<??> <object>” will be created as the parent of these features. The functionalities represented by the child features may together make up a larger functionality that is related to the object. The parent feature represents the larger functionality.

For example, two features contain object “gift certificate” in use case “Place an Order”, namely, “Enter gift certificate” and “Redeem gift certificate”. These two features compose a parent feature “?? gift certificate”, as shown in Figure 6(c).

- *D4: 'Discover decomposition relationships by identifying the “Is-a” relationship between objects’.*

If there exists an object that has a “Is-a” relationship with all the objects referenced by the names of a set of features, a feature named “<??> <object>” will be created as the parent of this set of features. The functionalities represented by the child features are special ways of realizing the parent feature.

For example, in use case “Place an order”, both “UPS” and “FedEx” are “shipping methods”. Features containing these two objects are “Choose UPS” and

“Choose FedEx”. These features compose a new parent feature “Choose the shipping method”, as shown in Figure 6(d).

- *D5: ‘Discover decomposition relationships by identifying ‘Has-a’ relationship between objects.’*

If there exists an object that has a “Has-a” relationship with all the objects referenced by the names of a set of features, a feature named “<??> <object>” will be created as the parent of this set of features. The functionalities represented by the child features may together realize the parent feature.

For example, in use case “Place an order”, “gift certificate” has “claim code” and “security code”. Features generated for these two objects compose a parent feature “Enter the gift certificate” as shown in Figure 6(e).

### 3. Characterization and Specialization relationship discovery rules:

- *CS1: ‘Discover characterization and specialization relationships by indentifying the method of an action.’*

If action *m* has a method, an attribute feature named “<??>” will be created. The relationship between the attribute feature and the feature generated from *m* is characterization. The relationship between the feature generated from the method of *m* and the attribute feature is specialization.

For example, the feature “Display shipping address error message” is characterized to the attribute feature ‘Display mode’. The attribute feature ‘Display mode’ is specialized to feature “by popping up a dialogue box”, as shown in Figure 6(f).

Figure 7 presents the algorithm for constructing AFMs. It first uses the “Is-a” and “Has-a” relationships between objects to build a set of object relation trees (ORTs) for each use case.

Having built the ORTs, a bottom-up strategy is adopted to build a feature model for each ORT. For each object in an ORT, rule *F1* and *F2* are first applied to analyze actions to identify features. Then the methods of the actions, sub flows linked to the actions and system operations following the actions are analyzed according to rule *CS1*, *D1* and *D2*, respectively. Then the features whose names reference the same object are united into a parent feature according to rule *D3*, and then *D4* and *D5* are applied to create the parent feature if the object is a part/kind of another object. When rule *D3*, *D4* and *D5* are applied to discover decomposition relationships, if all the names of the child features reference the same operation, “??” can be replaced by “<operation>”.

For each use case, if two or more feature models are constructed, we convert them into a single tree by adding an artificial root feature. We name the artificial root feature by referencing the name of the use case.

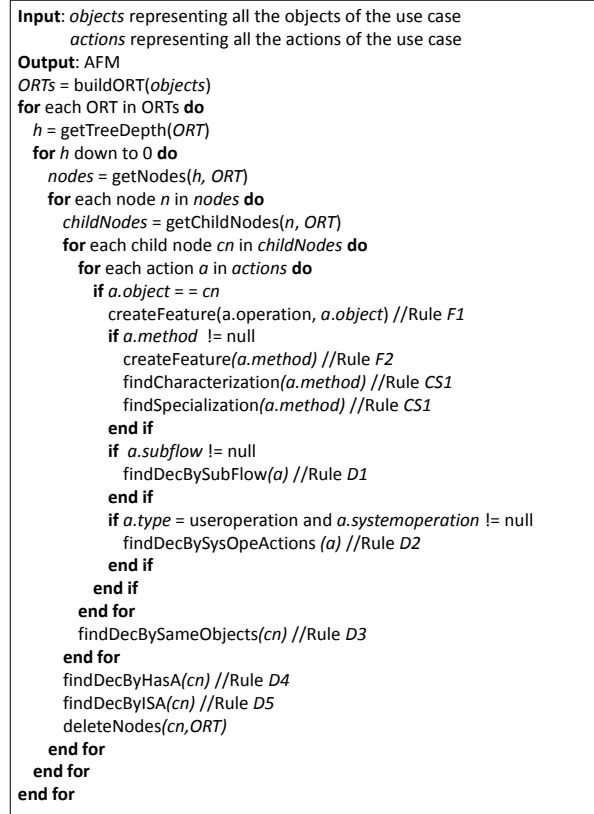


Figure 7. The algorithm for constructing AFMs

### 3.3. Adjust AFMs

After the AFMs have been constructed, they are checked and adjusted to eliminate improper features and relationships between features semi-automatically. Figure 8 depicts the process of adjusting AFMs.

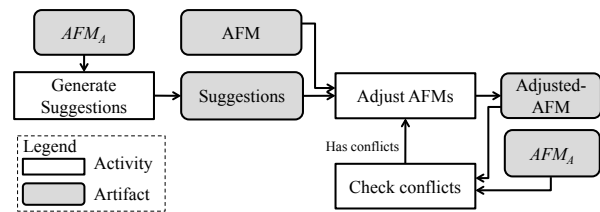


Figure 8. The process of adjusting AFMs

The AFMs are divided into two groups,  $AFM_U$  and  $AFM_A$ . The  $AFM_U$  contains all the unadjusted AFMs while the  $AFM_A$  contains all the adjusted AFMs. In our

approach, we adjust each AFM in the  $AFM_U$  and add the adjusted AFM into the  $AFM_A$ . When an AFM from the  $AFM_U$  is adjusted, the  $AFM_A$  are first analyzed to offer some adjusting suggestions, with the support of six heuristics adjusting rules. Then, three operations are provided to adjust the AFM. Finally, the newly adjusted AFM is added to the  $AFM_A$  if no conflicts are detected.

The adjusting operations provided in this paper are listed as follows:

- *Accept*

A feature is accepted if it really represents the characteristic of the application, and the accepted feature should be renamed if its name contains “?”. The accepted feature can also be renamed if its name is not satisfactory. For example, feature “Open the cart” is renamed by adding “the” between “open” and “cart”. A relationship between features is accepted if it really represents the relationship between features.

- *Reject*

A feature is rejected if it does not reflect the characteristic of the application. A relationship between features is rejected if it does not represent the relationship between features.

- *Create*

Both new features and new relationships between features can be created. To reduce the risk of introducing improper features and relationships, the items allowed to be created are:

- parent features that can be decomposed into several accepted features;
- relationships between two accepted features;
- *require* constraints between two accepted features.

Suggestions are generated with the help of six heuristics adjusting rules, as shown in Table 2. The circles labeled “A”, “R” or “C” indicate that the features/relationships are accepted, rejected or created. The suggestions can be declined if they are not satisfactory.

Before the newly adjusted AFM is added into the  $AFM_A$ , four conflicts checking rules are applied to detect the potential conflicts between them. The rules are listed as follows:

- If a feature/relationship is accepted in the  $AFM_A$ , it should be accepted in the newly adjusted AFM.
- If a feature/relationship is rejected in the  $AFM_A$ , it should be rejected in the newly adjusted AFM.
- In different feature models, the same child features should compose the same parent feature.
- If feature  $B$  is an ancestor node of feature  $A$  in some

Adjusted-AFM, the length of the paths from  $B$  to  $A$  in these Adjusted-AFM should be the same.

**Table 2. Rules for adjusting suggestion**

AFM <sub>i</sub> (to be adjusted)	AFM <sub>A</sub>	Adjusted-AFM <sub>i</sub>
	 Feature A is accepted.	 Suggest that feature A be accepted.
	 Feature A is rejected.	 Suggest that feature A be rejected.
	 Relationship between A and B is accepted.	 Suggest that relationship between A and B be accepted.
	 Relationship between A and B is rejected.	 Suggest that relationship between A and B be rejected.
	 Relationship between A and B is created.	 Suggest that relationship between A and B be created.
	 Feature A and relationships between A and B <sub>i</sub> are created.	 Suggest that feature A and relationships between A and B <sub>i</sub> be created.

Due to the page limitation, we only give an informal description about conflicts checking process. We apply conflicts checking rule 1, 2 and 3 when adjustments are made to an AFM. When adjusting suggestions are turned down or features are renamed without corresponding changes in the  $AFM_A$ , there are conflicts between the newly adjusted AFM and the  $AFM_A$ . To resolve the conflicts, the AFM or the Adjusted-AFM in the  $AFM_A$  should be adjusted again. To improve the efficiency of the whole approach, we apply conflicts checking rule 4 when merging the Adjusted-AFM into a DFM (See section 3.4).

### 3.4. Merge Adjusted-AFMs

After the AFMs have been adjusted, Adjusted-AFMs derived from the same use case set are merged into a DFM automatically.

Our algorithm for merging Adjusted-AFMs is based on our previous work [2], in which an Adjusted-AFM is first selected as a DFM. Then other Adjusted-AFMs are merged into the DFM one by one. The child nodes of a common node between DFM and Adjusted-AFM

are compared. If a node in the child nodes of the Adjusted-AFM does not exist in DFM, the node and its sub tree will be added to the DFM. If a child node exists in both the Adjusted-AFM and the DFM, it is identified as a new common node. In the algorithm, we set the root features of all the Adjusted-AFM as identical name and treat them as initial common nodes, because the corresponding use cases to the Adjusted-AFM describe similar functions. After all the Adjusted-AFM have been merged, the optionality of each feature is set according to the occurrence of the features in all the Adjusted-AFM.

```

Input: node  $u$  representing the common node in DFM and Adjusted-AFM
        node  $v$  representing the node to be added to DFM
        DFM
Output: isConflict
isConflict = false
dfmChildNodes = getChildNodes( $u$ ,DFM)
for each dfmChildNode in dfmChildNodes do
  subtree = getSubTree(dfmChildNode,DFM)
  if  $v$  exist in subtree
    isConflict = true;
  end if
end for

```

**Figure 9. The algorithm for checking conflicts**

```

Input: Directed Graph  $G1$ 
        (representing the features and constraints in the DFM)
        Directed Graph  $G2$ 
        (representing the features and constraints in the Adjusted-AFM)
Output:  $G1$ 
for each vertex  $u$  in  $G2$   $G1$  do
  for each vertex  $v$  required by  $u$  in  $G1$  do
    if !hasVertex( $v$ , $G2$ ) and !hasDirectedEdge( $v$ , $u$ , $G1$ )
      deleteDirectedEdge( $u$ , $v$ )
    end if
    if  $v$  exists in  $G2$ 
      if !hasDirectedEdge( $v$ , $u$ , $G1$ ) and !hasDirectedEdge( $u$ , $v$ , $G2$ )
        deleteDirectedEdge( $u$ , $v$ , $G1$ )
      end if
      if !hasDirectedEdge( $v$ , $u$ , $G1$ ) and hasDirectedEdge( $v$ , $u$ , $G2$ )
        deleteDirectedEdge( $u$ , $v$ , $G1$ )
      end if
      if hasDirectedEdge( $v$ , $u$ , $G1$ ) and !hasDirectedEdge( $u$ , $v$ , $G2$ )
        and !hasDirectedEdge( $u$ , $v$ , $G1$ )
        deleteDirectedEdge( $u$ , $v$ , $G1$ )
        deleteDirectedEdge( $v$ , $u$ , $G1$ )
      end if
    end if
  end for
end for
for each vertex  $m$  in  $G2$ - $G1$  do
  addVertex( $m$ , $G1$ )
end for
for each edge  $\langle m,n \rangle$  in  $G2$  do
  if  $m$  lexist in  $G1$  or  $n$  lexist in  $G1$ 
    addDirectedEdge( $m$ , $n$ , $G1$ )
  end if
end for

```

**Figure 10. The algorithm for merging constraints**

We use the same idea to merge the Adjusted-AFM. Furthermore, our method can detect conflicts between Adjusted-AFM and merge constraints. We provide two algorithms for checking conflicts and merging constraints, as depicted in Figure 9 and Figure 10, respectively.

During the process of checking conflicts, when a node  $v$  and its sub tree are added to the DFM, the algorithm finds all the child nodes of the common node  $u$  in the DFM. If node  $v$  exists in the sub trees of these child nodes, there is a conflict among the merged Adjusted-AFM (Conflicts checking rule 4).

The *require* constraints between features are also merged when the Adjusted-AFM are merged. We adopt the twelve constraints merging rules in [7] to merge constraints. A directed graph can be generated from each Adjusted-AFM. The features are the vertexes while the binary *require* constraint between features is a directed edge. When two Adjusted-AFM are merged, directed graphs from each model are also merged. The algorithm of the constraints merging is depicted in Figure 10.

## 4. Case Study

In this section, we study web store systems to demonstrate our approach. In this study, for simplification, the input use case set contains three use cases that are chosen from three web store systems: www.newegg.com (System  $A$ ), www.amazon.com (System  $B$ ), www-ssl.bestbuy.com (System  $C$ ).

**Table 3. The “Is-a” relationships between objects for system A**

Parent object	Child objects
Payment method	Credit Card, PayPal, BillMeLater, Mail Payment, Promo code, Gift Certificate, Gift Card, Reward Certificate, Checking account
Shipping method	DHL, FedEx, UPS

**Table 4. The “Has-a” relationships between objects for system A**

Parent Object	Child objects
Order	Product, Shipping address, Payment method, Shipping method, Total Price
Product	Quantity
Shipping address	First Name, MI, Last Name, City, Address, Zip Code, Phone, State
Gift certificate	Claim code, Security Code

All of the three use cases describe how to place an order in a web store. For conciseness, only the use case

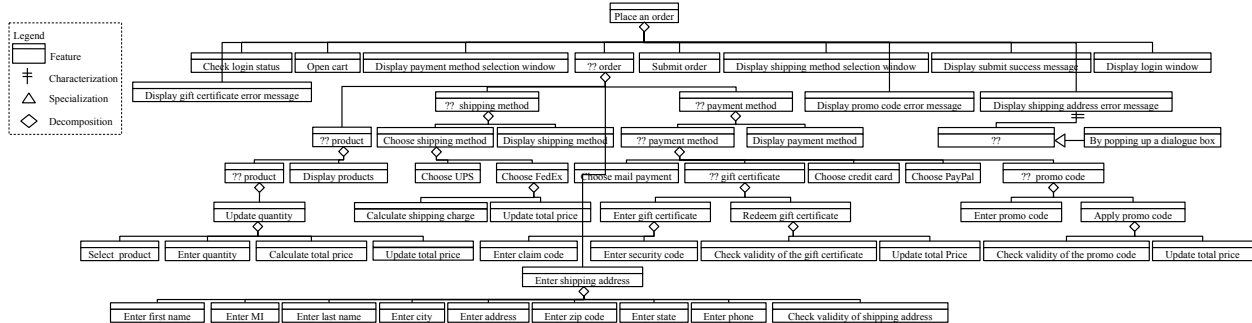


Figure 12. An AFM of system A

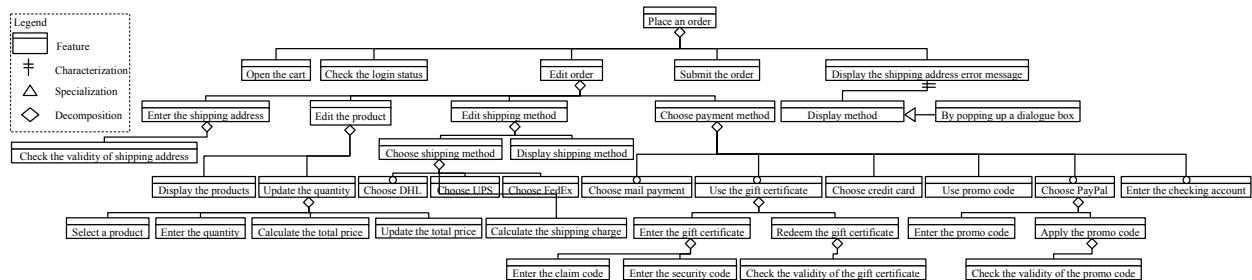


Figure 13. A DFM of the web store domain

from system A is presented in this paper (Table 1).

First, for each use case, the “Is-a” and “Has-a” relationships between the objects are identified. Table 3 and Table 4 show the two relationships from system A. Similar operations and similar objects in all the three use cases are assigned with identical names, as shown in Table 5 and Table 6.

Table 5. Similar operations in all the use cases

Identical name	Operations with similar semantics
Enter	Enter, Input
Use	Use, Redeem, Apply
Display	Prompt, Display

Table 6. Similar objects in all the use cases

Identical name	Objects with similar semantics
Shipping cart	Shipping cart, Cart
Payment method	Payment method, Payment option
Shipping charge	Shipping charge, Delivery charge
Promo code	Promo code, Promotional Codes, Promotional claim codes

Second, a set of ORTs are built for each use case. Figure 11 shows the ORTs generated from system A (13 single node trees and a tree whose height is 3). Those ORTs are used to build an AFM. Here we only choose system A to illustrate the construction of AFMs

due to the space, as shown in Figure 12. This AFM represents the characteristics relating to “Place an order” of system A.

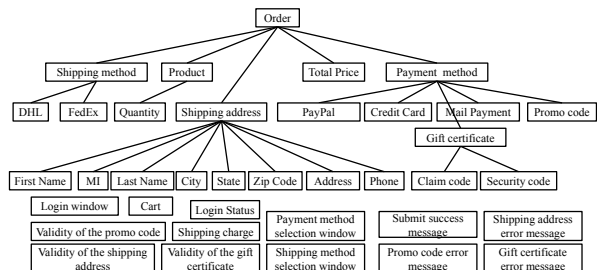


Figure 11. The ORTs for system A

Third, the AFMs are checked and adjusted to eliminate improper features and relationships. Also, accepted features with improper/undetermined names are renamed.

Finally, the Adjusted-AFM is merged into a DFM, as shown in Figure 13. This DFM represents the characteristics relating to “Place an order” of the web store domain.

## 5. Related work

In FODA [4] and FORM [5], feature models are constructed by analyzing a set of applications in a

specific domain in terms of services, operating environments, domain technologies, and implementation techniques. In FeaturSEB [3], a domain use case model is derived from individual use cases. Then a feature model is extracted from the domain use case model. However, all of the three methods pay little attention to the automatic construction of feature models.

Braganca and Machado [1] propose an automatic approach to the transformation between feature models and use case models. Their approach explores the *include* and *extend* relationships between use cases to discover relationships between features. We differ from their work because we explore the inner description of use cases to discover and organize features.

In our previous work [2], we propose a semi-automatic approach to the construction of feature models based on requirements clustering. In this approach, feature models are derived from requirements specifications, while the approach proposed in this paper focuses on how to construct feature models from use cases.

## 6. Conclusion and Future Work

This paper presents a semi-automatic approach for constructing feature models from use cases, with the purpose to minimize analysts' participation in the activities of feature-oriented commonality and variability modeling. In our approach, inner descriptions of use cases are automatically analyzed to generate AFMs. Then, these AFMs are checked and adjusted to eliminate improper features and relationships. After the AFMs have been adjusted, they are merged automatically into a DFM that represents partial characteristics of the domain. A set of rules and algorithms are proposed to support the process of feature models' construction.

Our future work will focus on how to integrate the partial DFMs to represent the overall characteristics of the domain. We plan to explore the relationships between use cases (such as *include*, *exclude*) deeply to establish the relationships between DFMs. We will also continually investigate the applicability of this approach.

## Acknowledgments

This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2009CB320701, the Science Fund for Creative Research Groups of China under Grant No. 60821003, the Hi-Tech Research and Development

Program of China under Grant No. 2006AA01Z156, and the Natural Science Foundation of China under Grant No. 60703065 and 60873059.

## References

- [1] A. Braganca, R.J. Machado, "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines", 11th International Software Product Line Conference (SPLC 2007), Sept. 2007, pp. 3-12.
- [2] K. Chen, W. Zhang, H. Zhao, H. Mei, "An approach to constructing feature models based on requirements clustering", In *Proceedings of 13th IEEE International Conference Requirements Engineering*, 2005, pp. 31-40.
- [3] M.L. Griss, J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB", In *Proceedings of Fifth International Conference on Software Reuse*, IEEE Computer Society, Canada, Jun. 1998, pp. 76-85.
- [4] K. C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
- [5] 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*, 1998, 5:143-168.
- [6] H. Mei, W. Zhang, and F. Gu, "A Feature Oriented Approach to Modeling and Reusing Requirements of Software Product Lines", In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC 2003)*, IEEE Computer Society Press, Dallas, USA, November 2003, pp. 250-255.
- [7] Sergio Segura, David Benavides, and Pablo Trinidad, *Generative and Transformational Techniques in Software Engineering II*, Springer Berlin, 2008, pp. 489-505.
- [8] G.Winters, "Use Case Terminology", *IEEE Software*, vol.22, no.2, March-April 2005, pp. 67-67.
- [9] W. Zhang, H. Zhao, and H. Mei, "A Propositional Logic Based Method for Verification of Feature Models", In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 2004)*, Springer-Verlag, volume 3308 of Lecture Notes in Computer Science, Seattle, WA, USA, November 2004, pp. 115-130.
- [10] W. Zhang, H. Mei, and H. Zhao, "A feature-oriented approach to modeling requirements dependencies", In *Proceedings of 13th IEEE International Conference on Requirements Engineering*, 2005, pp. 273-282.