An Approach to Constructing Feature Models Based on Requirements Clustering

Kun Chen, Wei Zhang, Haiyan Zhao, Hong Mei
Institute of Software, School of Electronics Engineering and Computer Science, Peking University,
Beijing, 100871, P. R. China
{chenkun, zhangw, zhhy}@sei.pku.edu.cn, meih@pku.edu.cn

Abstract

Feature models have been widely adopted in software reuse to organize the requirements of a set of similar applications in a software domain/product-line. However, in most feature-oriented methods, the construction of feature models heavily depends on the domain analysts' personal understanding, and the work of constructing feature models from the original requirements of sample applications is often tedious and ineffective.

This paper proposes a semi-automatic approach to constructing feature models based on requirements clustering, which automates the activities of feature identification, organization and variability modeling to a great extent. The underlying idea of this approach is to analyze the relationships between individual requirements and cluster tight-related requirements into features. With the automatic support of this approach, good-quality feature models can be constructed in a more effective way. A case study is also provided to show the feasibility of this approach.

1. Introduction

Software reuse has long been recognized as an effective way to improve software quality and productivity [17]. A feasible way of software reuse is to produce general purpose software artifacts, followed by customization to accommodate different situations.

Feature-oriented domain analysis methods [11, 12, 7, 2] have been proposed to facilitate the customization of software requirements. These methods treat features as the first-class entities in the requirements level. In the domain analysis, features and relationships between features (called domain feature model) are used to organize the requirements of a set of similar applications in a software domain/product-line. Then, in the development of specific applications, the domain feature model is customized into application feature models according to specific reuse contexts.

However, few feature-oriented methods pay attention to adopting automated techniques for the construction of feature models. The construction process often becomes ineffective since it involves systematic analysis of a set of sample applications. Moreover, without automatic support, the quality of resultant feature models heavily depends on the analysts' personal experience and understanding.

In this paper, we propose a requirements clustering-based approach to constructing feature models from the functional requirements of sample applications. The purpose of our approach is to automate the construction process as much as possible. In this approach, sample applications are first analyzed, and a set of corresponding application feature models are built. Then these application feature models are merged into a domain feature model, and variable features are also labeled based on the difference analysis of these application feature models. With the automatic support of this approach, good-quality feature models can be constructed in a more effective way.

The rest of this paper is organized as follows. Section 2 introduces some preliminaries to our approach. Section 3 describes this approach in detail. Section 4 evaluates this approach from two aspects. Section 5 demonstrates the feasibility of the approach through a case study. Related work is discussed in Section 6. Finally, Section 7 summarizes this paper and outlines the future work.

2. A metamodel of feature models

In this section, we present a metamodel of feature models based on our previous work [16, 25].

The metamodel of feature models is shown in Figure 1, which consists of three basic concepts: Feature, Refinement, and Constraint. All these concepts are subclasses of another two concepts: Classifier and Relationship in UML Core Package [20].

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE

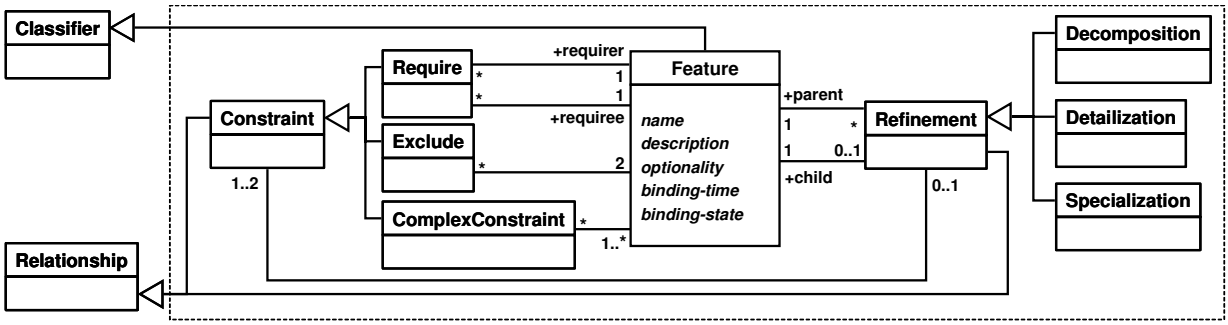

Figure 1. The metamodel of feature models

A feature describes a product characteristic from user or customer views [12], which essentially consists of a cohesive set of individual requirements [21, 24, 15]. In this metamodel, a feature contains the following attributes: *name*, *description*, *optionality*, *binding-time* and *binding-state*.

Name is the denotation of a feature, and it can be all meaningful character strings. *Description* is a detailed representation of a feature. *Optionality* indicates whether a feature has the chance to be removed from the current feature model when its parent feature (if has) has been bound. This attribute has two values: *mandatory* and *optional*. *Binding-time* is an attribute related to optional features. It describes a phase in the software life-cycle when an optional feature should either be bound or removed from the current feature model. Typical binding-times include *reuse-time*, *compile-time*, *deploy-time*, *load-time*, and *run-time*. The attribute *binding-state* describes whether a feature is *bound*, or *removed*, or *undecided*.

In the metamodel, *refinements* between features are classified into three categories: *decomposition*, *detailization* and *specialization*. Refining a feature into its constituent features is called *decomposition* [11]. For example, the feature *edit* in many software systems is often decomposed into three sub-features: *copy*, *paste* and *delete* (Figure 2). Refining a feature by identifying its attribute features is called *detailization*. For example, in graph-editor domain, the feature *graph-move* can be detailized by two attribute features: *moving-mode* and *moving-constraint*. Refining a general feature into a feature incorporating further details is called *specialization* [11]. For instance, the feature *moving-mode* in graph-editor domain has two specialized features: *outline-moving* and *content-moving*. *Specialization* is often used to represent a set of variants of a general feature in feature models. The general feature is also called *variation point feature* [7].

Features with different abstract levels and granularities are organized into a hierarchy structure through refinement relationships between them. More strictly, this metamodel

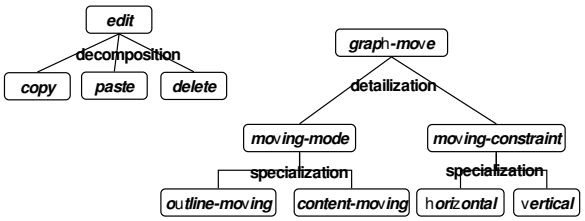

Figure 2. Example of refinements

limits this hierarchy structure to be one or more feature trees. That is, a feature is either a root feature or refined exactly from one feature. This limitation contributes to the simplicity and understandability of feature models. In a feature model, if there are two or more trees, we can convert them into a single feature tree by adding an *artificial root node* as the parent node of their root nodes. This artificial root node is also called *concept node* [4].

If two features have the same parent feature, we define that they are in the same *cluster* or *group*.

A *constraint* captures a set of rules that must be satisfied when customizing feature models. Only the customization results that don't violate constraints on features are the candidates of valid application feature models. [11] defines two kinds of binary constraint on features: *require* and *exclude*. For features A and B, "A *require* B" indicates the rule that B cannot be removed from the current feature model when A is not removed. "A *exclude* B" means that at most one of the two features can be in the bound state in a same context. Other kinds of constraints involving a set of features are called *ComplexConstraints*.

3. Construction of feature models

In this section, we describe an approach to constructing feature models, which is based on requirements clustering. The overall process of this approach is depicted in Figure 3. There are two main stages in this process: construction of

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE

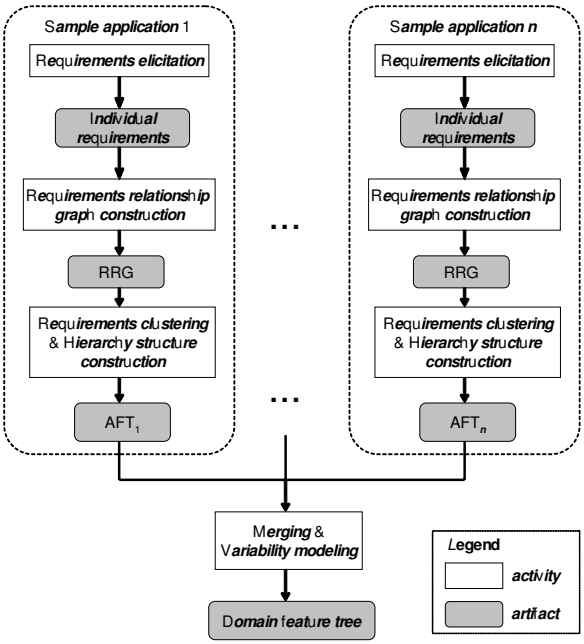

Figure 3. The overall process of feature model construction

application feature trees (AFTs) and construction of domain feature tree (DFT).

In the first stage, several sample applications are analyzed. For each of them there are three activities. The first activity is to elicit functional requirements for the sample application. Many systematic techniques [3] have given enough support for this activity. We assume each of the functional requirements is documented as long as it clearly specifies a certain behavior of the system. These requirements are called individual requirements in this paper. The second activity is to model individual requirements and the relationships between them in an undirected graph, which is called requirements relationship graph (RRG). This activity will be stated in Section 3.2. The third activity is to identify and organize features by applying the clustering algorithm in RRG. The underlying idea is that a feature is a cluster of tight-related requirements, and features with different granularities can be generated by changing the clustering threshold value (see Section 3.3). After finishing these three activities, we get an application feature tree for the sample application.

The second stage is to merge the application feature trees of all the sample applications into a single domain feature tree, and label the variable features. We will describe this stage in Section 3.4.

In our approach, most of these activities can be automatically processed. On the other hand, this approach may need human intervention according to the domain knowledge. That is, the AFT of each sample application may need to be adjusted before the merging activity, and the DFT may need to be adjusted after the merging activity.

3.1. Classification of relationships between requirements

Before we describe each activity in detail, we introduce a classification of relationships between requirements, which is essential for the quantification of the relationships. For analyzing the relationships, we also introduce the concept of *resource*, which stores the data accessed (modified or read) by requirements to behave correctly. One example of resource in email-client software is *received-box*, which stores the received emails.

Based on our study and other researchers' work [1, 9], we define five kinds of relationships which are helpful to the clustering activity. They are listed as follows, in which we take the email-client software domain for example. These classifications are called basic relationships between requirements in this paper.

- *Strong Resource Relationships* ($resource_s$). For any two requirements A and B, if both of them modify the same resource, there is a strong resource relationship between A and B. For example, both of the requirements *email-writing* and *attachment-adding* modify the resource *email*, and they have a strong resource relationship.

- *Weak Resource Relationships* ($resource_w$). For any two requirements A and B, if they access the same resource, and at least one of them read this resource, there is a weak resource relationship between A and B. For example, both of the requirements *email-viewing* and *email-searching-by-title* read the resource *email*, and they have a weak resource relationship.

- *Necessity Require Relationships* ($require_n$). For any two requirements A and B, A $require_n$ B means that for A to behave correctly, it will depend on whether B can behave correctly. One example is "*email-forwarding* $require_n$ *email-sending*".

- *Availability Require Relationships* ($require_a$). For any two requirements A and B, A $require_a$ B means that whether A can behave depends on whether B have behaved. That is, if B hasn't behaved, A will never be available to users. Those requirements whose behavior is triggered by other requirements often involve such relationships. One example is "*email-copy-auto-saving* $require_a$ *email-sending*", in which the function of *email-copy-auto-saving* is to save the copy of a sent-out email to copy-box. And only when an email

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE

has sent out, the behavior of requirement *email-copy-auto-saving* is triggered.

- *Or Relationships* (*or*). This kind of relationships exits between requirements which specify quite similar behaviors, except that some details of the behaviors are different, such as the condition for behavior, the input value, and so on. One example is both of the requirements *email-searching-by-sender* and *email-searching-by-title* specify the behavior of searching emails, and the only difference between them is the condition for searching.

There is another kind of relationships called *exclude*, which indicates two requirements are conflicting when behaving simultaneously. Two such requirements may be in different clusters; they also may be in the same group, when both of them are specialized features of a same parent feature and at most one of them is allowed to be bound at the same time. So we find exclude relationships contribute little to the clustering activity.

3.2. Build the requirements relationship graph

After individual requirements are elicited for a sample application, the requirements relationship graph is built. Assume there are n requirements, they and their relationships are modeled as an undirected graph $G = (V, E)$, in which $V = \{R_i \mid R_i$ is an individual requirement, $1 \leq i \leq n\}$, and $E = \{E_{ij} \mid E_{ij}$ is the relationship between requirements R_i and $R_j, 1 \leq i, j \leq n\}$.

The key point is to determine the weight of each edge E_{ij} to express the strength of the relationships between requirements R_i and R_j. As we recognize that there may be several basic relationships between two requirements, we adopt the following quantification strategy.

First, we define the weight function $W(t)$ for each basic relationship, where $t \in T$, $T = \{resource_s, resource_w, require_n, require_a, or\}$. This function maps each basic relationship to a positive integer, which is denoted by a symbol as described in Table 1. The values of W_{sr}, W_{wr}, W_{nr}, W_{ar} and W_{or} are determined by analysts according to the specific domain. These values have the following order relations: W_{or} is the largest of them. This is because the *or* relationship exists between quite similar requirements, and its weight should express a very strong cohesion. W_{sr} is larger than W_{wr}, since $resource_s$ is stronger than $resource_w$. For the similar reason, W_{nr} is larger than W_{ar}.

Then, we define the weight function for each edge. The weight of the edge between two requirements is the sum of the weights of their basic relationships. That is, the weight function $W(E_{ij})$ for each edge is $W(E_{ij}) = W(t_k)$, where there exists a basic relationship t_k between the requirements R_i and R_j.

Table 1. The symbol for the weight of each basic relationship

Basic Relationship	Symbol for Weight
Strong Resource	W_{sr}
Weak Resource	W_{wr}
Necessity Require	W_{nr}
Availability Require	W_{ar}
Or	W_{or}

Though there are some costs in the construction of RRG (because we need to examine at most $O(n^2)$ relationships for n requirements), we can get high benefits from RRG in the following aspects. First, once the RRG for a specific application is constructed, it can be reused for other applications in the same domain. Second, the RRG provides a base for the subsequent activities in our work, which can be automated to a great extent. Third, the domain knowledge embodied in RRG can help analyze both static and dynamic dependencies between features.

3.3. Cluster requirements into features

After building the requirements relationship graph, we apply requirements clustering in this graph to identify and organize features. The underlying idea is that a feature is a cluster of tight-related requirements, and features with different granularities can be generated by changing the clustering threshold value.

The algorithm for requirements clustering is described in Algorithm 1. In this algorithm, the attribute *validType* of each edge indicates whether this edge is valid for computing the connected components. And the functions used in the algorithm are defined below.

- **distinctEdgeWeightsNum(**G**)** returns the number of distinct edge weights in G.
- **sortDistinctEdgeWeights(**G**)** is used to sort all the distinct edge weights in the increasing order. This function returns an array of sorted edge weights.
- **addToLevel(**AFT**,** i**,** c**)** is used to add a node representing the cluster c to the level i of the tree AFT.
- **connectedComponentsByValidEdges(**G**)** is used to decompose the graph G into connected components. *Any vertices in the same connected component are reachable from each other through the edges whose validType attribute is true*. This function returns a set of connected components, and each of them represents

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE

a feature (cluster) consisting of a set of tight-related requirements.

- **addRefinement(C_i, C_j)** is used to add a refinement relationship between C_i and C_j. And C_i is the parent feature of C_j.
- **unite(C_i, C_j)** is used to unite features C_i and C_j into a single feature.

Algorithm 1 Requirements clustering

Input: undirected graph G representing a requirements relationship graph; t as the threshold for clustering
Output: the application feature tree AFT
{step 1: initialization}
$AFT = \phi$
$m =$ distinctEdgeWeightsNum(G)
$S =$ sortDistinctEdgeWeights(G)
for each node v in G **do**
  addToLevel(AFT, $m+1$, v)
end for
{step 2: feature identification by clustering}
for $i = m$ downto 1 **do**
  $t = S[i]$
  for each edge e in G **do**
    if $W(e) \geq t$ **then**
      $e.validType = true$
    else
      $e.validType = false$
    end if
  end for
  $C =$ connectedComponentsByValidEdges(G)
  for each cluster c in C **do**
    addToLevel(AFT, i, c)
  end for
end for
{step 3: hierarchy structure construction}
for $L = 1$ to m **do**
  for each cluster C_i in level L of AFT **do**
    for each cluster C_j in level $L+1$ of AFT **do**
      if $C_i \supset C_j$ **then**
        addRefinement(C_i, C_j)
      else if $C_i = C_j$ **then**
        UNITE(C_i, C_j)
      end if
    end for
  end for
end for

Now we explain each step of this algorithm in detail. Step 1 is initialization. All the individual requirements are placed in the lowest level of AFT, and they are the features with the finest granularity. Then, all the distinct edge weights (assume there are m distinct edge weights) of G are sorted, and each of them will be used in step 2 as the threshold value.

Step 2 is to identify features with different granularities by requirements clustering, consisting of m loops. In each loop, a threshold value t is fixed. If there is an edge between two requirements and its weight is greater than or equal to t, they *will* be put into the same cluster; if the weight is less than t, they *may* be in the different clusters. So the edges whose weights are above or equal to the threshold value are set to be valid; otherwise, the edges are invalid. Then connected components are computed by the valid edges. Each connected component is a cluster of tight-related requirements (i.e. a feature). In the end of each loop, the generated features are put at the corresponding level of AFT. As we decrease the threshold value, more edges are set to be valid, and we get features (clusters) with coarser granularities.

In step 3, the hierarchy structure is constructed. The features (clusters) in successive levels of the tree are explored. If a lower-level feature is a subset of another higher-level feature, we build the refinement relationship between them; if they are identical, we only reserve one of them. After all the levels are examined, all the refinement relationships are built.

The resultant hierarchy structure may need to be adjusted according to domain knowledge. The most important adjustment is to label the specialization relationships. If several requirements in the same cluster have *or* relationships, for each requirement, we should label the specialization relationship between it and its parent feature. Moreover, we may add features or remove features.

After the adjustment, we should examine whether there are two or more trees. If so, we add an *artificial root node* as the parent node of their root nodes. Thus, we get an application feature tree for a specific sample application. In the tree, each node is a feature, and all the features are organized by refinement relationships. But the tree contains no information about variability; how to model variability is the topic of Section 3.4.

3.4. Merging and variability modeling

As discussed before, the application feature tree only captures the requirements of a specific application. After the application feature trees are constructed for all the sample applications, we can construct the domain feature tree by merging these application feature trees and modeling their variability.

The algorithm for merging and variability modeling is described in Algorithm 2, which is partly inspired by the work of Wen et al. [23]. In this algorithm, each AFT has an identical artificial root node *rt*, and each feature has an attribute *occurrence*, which is used to record its occurring

time in all AFTs. The function DFS-Visit is a key function, and is also described in Algorithm 2. Other functions used in the algorithm are defined below.

Algorithm 2 Merging and variability modeling

Input: M trees AFT_1, AFT_2, ..., AFT_M, representing M application feature trees; rt as the identical artificial root node in each tree; t as the threshold for variability modeling
Output: the domain feature tree DFT
{step 1: merge application feature trees}
$DFT = AFT_1$
for each node v in DFT **do**
  $v.occurrence = 1$
end for
for $i = 2$ to M **do**
  AFT = AFTi
  DFS-Visit(rt)
end for
{step 2: set each feature's optionality}
for each node v in DFT **do**
  u = parentnode(v)
  if $v.occurence/u.occurence \geq t$ **then**
    $v.optionality = mandatory$
  else
    $v.optionality = optional$
  end if
end for
{function DFS-Visit is described as follows}
DFS-Visit(u)
$S1$ = subnodes(DFT, u)
$S2$ = subnodes(AFT, u)
if $S1 \neq \phi$ and $S2 \neq \phi$ **then**
  for each node v in $S2 - S1$ **do**
    for each node w in subtree under v **do**
      $w.occurence = 1$
    end for
    addSubtreeToDFT(u, v)
  end for
  for each node v in $S1 \cap S2$ **do**
    $v.occurence = v.occurence + 1$
    DFS-Visit(v)
  end for
end if

- **parentnode**(v) is used to return the parent node of *v* in *DFT*.
- **subnodes**($tree$, u) is used to return an array of all the direct subnodes of the node *u* in *tree*.
- **addSubtreeToDFT**(u, v) is used to add the subtree under node *v* (including *v*) in *AFT* to *DFT*, as the subtree under the node *u*. All the nodes' *occurrence* attributes and all the refinement relationships in the subtree are kept the same in *DFT*.

In step 1 of this algorithm, first we select an application feature tree as DFT, in which each node's $occurrence$ attribute is set to 1. Then there are $M - 1$ loops for merging application feature trees. In each loop, a next application feature tree is selected as AFT to merge with current DFT. In the merging, a depth-first searching strategy is used to compare the features in DFT with the features in AFT. In the beginning, the identical artificial root node in both trees is a comparison node. Then we compare the child-node sets of the comparison node in both trees. The comparison results are divided into three categories. First, if a node exists in DFT's child-node set but not in AFT's child-node set, this node and its subtree nodes will remain as before. Second, if a node exists in AFT's child-node set but not in DFT's child-node set, this node and its subtree nodes will be added to the DFT as new nodes, and each node's occurrence will be set to 1. Third, the nodes existing in both child-node sets are called common nodes, and each node's occurrence is added by 1. Each common node will be a new comparison node and the algorithm will deal with all the common nodes recursively.

The second step is to set each feature's $optionality$ attribute by its $occurrence$. Recalling the definition of $optionality$ in Section 2, we recognize that each feature's $optionality$ is pertinent to its parent. So we decide each feature's $optionality$ by its $occurrence$ divided by its parent's $occurrence$. Obviously, this quotient is ranged from 0 to 1. If the quotient is larger than the threshold value t (it is chosen according to the domain knowledge), the feature's $optionality$ is set to *mandatory*; otherwise, it is an *optional* feature. In particular, the identical artificial root node exists in each AFT and has no parent node. We define that its $occurrence$ is the total number of merged application feature trees, and its $optionality$ is always *mandatory*.

4. Evaluation

In this section, we evaluate our approach from two aspects, namely the time complexity and the quality of generated clusters. The first aspect measures the cost of our approach, and the second one reflects the effect of our approach.

4.1. Time complexity

In Algorithm 1, let $G = (V, E)$ be the requirements relationship graph, with $|V| = n$, and assume there are m distinct edge weights. In each loop of step 2, the computation of connected components can be finished in time

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE

$O(|V| + |E|) = O(n^2)$. As there are at most m loops, the total time of feature identification is $O(mn^2)$. In each loop of step 3, there are at most n clusters in a level. If comparing two clusters takes time $O(n)$ with proper implementation, each loop takes time $O(n^3)$. So the hierarchy construction takes time $O(mn^3)$.

In Algorithm 2, assume there are M sample applications, and the node number of the final domain feature tree is N. There are $M - 1$ loops in step 1, and each loop involves a depth-first search in trees. Thus, the upper time bound of merging and variability modeling is $O(MN^2)$ with proper implementation.

In summary, the time complexity of automated activities of this approach is $O(mn^3 + MN^2)$.

4.2. Cluster quality

To evaluate the quality of generated clusters, we use the independency metric (IM) [14]. It is originally used to evaluate the results of component identification, and we find it's also useful for our approach.

IM calculates the quality for a given subgraph (cluster) $C = (V', E')$ of a graph $G = (V, E)$. The set of outgoing edges of C is $O = \{< u, v > | < u, v > \in E, u \in V', v \in V - V'\}$. Then the independency metric for C is defined as follows [14].

$$IM(C) = \frac{\sum_{e \in E'} W(e)}{| V' | \sum_{e \in O'} W(e)}$$

In this formula, the numerator is the sum of the weight of each edge in the subgraph C, and it represents the cohesion of C. The first denominator, $|V'|$, denotes the size of C. Another denominator is the sum of the weight of each outgoing edge, and it represents the coupling between C and other clusters. These three factors altogether define the independency metric for a cluster. The IM value is larger, the independency of a cluster is higher. That is, its quality is higher.

In our approach, the IM values are computed for specific systems. And we will use them in Section 5.

5. Case study: library management systems

In this section, we study the library management systems in data-dominant domains. A data-dominant system is one which primarily concerns the integrity of the system data [8]. In this study, for simplification, we only choose two similar sample applications. We use the proposed approach to construct feature models for them, and then evaluate our approach from two aspects presented in Section 4.

Table 2. Individual functional requirements of a library management system

R1.	Lend a copy of a book.
R2.	Return a copy of a book.
R3.	Renew a copy of a book.
R4.	Reserve a book.
R5.	Cancel a reservation for a book.
R6.	Add a user of the library.
R7.	Remove a user of the library.
R8.	Modify the information of a user.
R9.	Add a copy of a book to the library.
R10.	Remove a copy of a book from the library.
R11.	Remove all copies of a book from the library.
R12.	Get a list of books in the library by title.
R13.	Get a list of books in the library by author.
R14.	Get a list of books in the library by publisher.
R15.	Find out what books are currently checked out by a user.
R16.	Get a list of users by address.
R17.	Get a list of users by name.
R18.	Find out all the reservations made by a user.
R19.	Remind a user his/her lent book is overdue.
R20.	When a book is overdue, automatically remind the user who has lent the book.
R21.	Find out the total number of library users.

The individual functional requirements of the first sample application are listed in Table 2. These requirements are adapted from the case study in [9].

First, we analyze the relationships between these requirements. For conciseness, we describe the following examples to show how to identify the basic relationships. Other basic relationships can be analyzed in the same way.

Four kinds of resource are identified in this system: *checkout lists*, *reservation lists*, *users*, and *books*. The requirements R1 and R2 are to modify the resource *checkout lists*, while R15 is to read that resource. So, *strong resource relationship* exists between R1 and R2, and *weak resource relationships* exist between R1 and R15, R2 and R15.

The requirements R20 and R19 have a *necessity require relationship* because the correct behavior of automatic reminding depends on the behavior of reminding.

The requirements R1 and R2 have an *availability require relationship* because the behavior of returning a book is not available until at least a book is checked out.

The requirements R16 and R17 have an *or relationship*, because they state the similar behavior of searching users, the only difference is search condition.

Then the RRG for this system is constructed (Figure 4). The weights of basic relationships are chosen as $W_{sr} = 2$, $W_{wr} = 1$, $W_{nr} = 2$, $W_{ar} = 1$, $W_{or} = 2$ in this study.

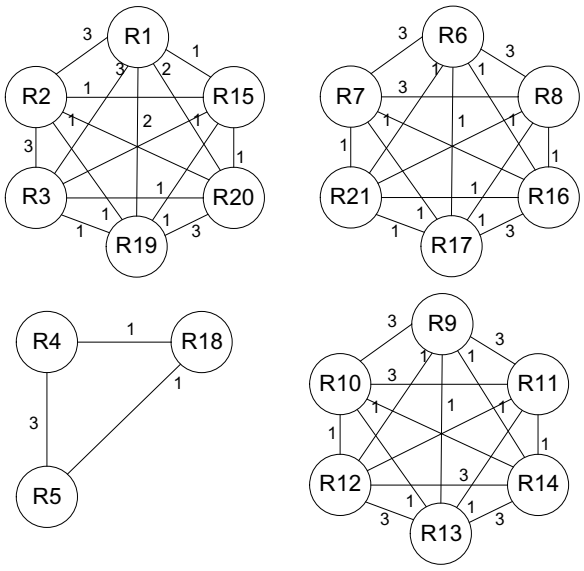

Figure 4. Requirements relationship graph for the first sample application

Afterwards, we apply the requirements clustering algorithm. There are three distinct edge weights: 1, 2, and 3. At first, individual requirements are put at the lowest level of the tree. Then as the threshold value changes from 3 to 2, then to 1, requirements are clustered into features. For example, when threshold is 3, we can get 10 clusters: the first seven clusters in Table 3, plus {R15}, {R18} and {R21}.

After the construction of hierarchy structure and the adjustment for understandability, we get the AFT as depicted in Figure 5, in which all the features are given a meaningful name. In this figure, the artificial root node is colored in grey, and the specialization relationships are also labeled.

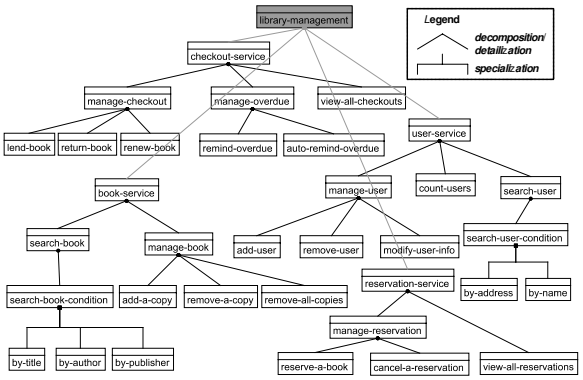

Figure 5. Application feature tree for the first sample application

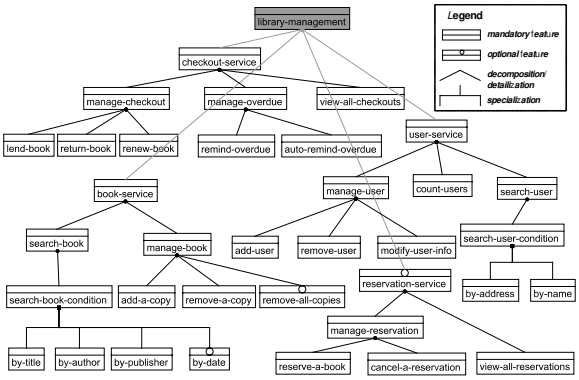

Figure 6. Domain feature tree for library management systems

The second sample application is similar to the first one except for the following difference. The requirements R4, R5, R18 and R11 are not contained in this system. In addition, a new requirement is added, which states the behavior of searching books by the publishing date. The AFT for this system is constructed in the same way. Then, it is merged with the AFT in Figure 5. When modeling variability we choose the threshold value as 1. After these activities, we get the DFT for library management systems, as demonstrated in Figure 6, with the mandatory and optional features labeled.

Finally, we evaluate the cost and the effect of this approach as described in Section 4. First, because the selected systems are relatively small-scale and the automated activities have polynomial time complexity, the cost of construction process is expected to be much less than manual construction. Second, we evaluate the quality of generated clusters using independency metric. For conciseness, only the IM values for most features (clusters) of the first sample application are listed, as shown in Table 3. In this table, the last four clusters have no outgoing edges, so their IM values are not computed. From the results, we can see each cluster has a relatively high IM value when compared with all the other possible subgraphs which have the same size with this cluster. That is, each of the generated features (clusters) has a good quality. For example, the IM value of the cluster *manage-book* is 0.33, which is the largest IM value among the subgraphs of size 3.

6. Related work

Since the concept of feature model was introduced in FODA [11], feature-oriented domain analysis methods have been used in domain engineering and product line engineering for many years. In these methods, feature models

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE

Table 3. Features (clusters) in the first sample application

Feature (cluster) name	Constituent requirements	Size	IM
manage-checkout	R1, R2, R3	3	0.27
manage-user	R6, R7, R8	3	0.33
manage-reservation	R4, R5	2	0.75
manage-book	R9, R10, R11	3	0.33
manage-overdue	R19, R20	2	0.15
search-user-condition	R16, R17	2	0.19
search-book-condition	R12, R13, R14	3	0.33
checkout-service	R1, R2, R3, R15, R19, R20	6	/
book-service	R9, R10, R11, R12, R13, R14	6	/
user-service	R6, R7, R8, R16, R17, R21	6	/
reservation-service	R4, R5, R18	3	/

are at the center of the domain requirements models.

FODA [11] and FORM [12] provide some principles for constructing feature models, including the principles for feature identification, classification and organization. FeatureRSEB [7] integrates the feature modeling process of FODA with the RSEB (Reuse-Driven Software Engineering Business) method. It uses feature models to represent the variabilities of the use-case models and object models. It also describes a process of feature model construction, which assumes that a feature is only traced to one use case, and the features are extracting from the use cases. Generative programming [4] also uses the feature models for reuse in product families, and it provides a brief guideline for feature modeling.

All the above methods concentrate on the content and the utilization of feature models. But they pay relatively little attention to adopting automatic techniques for constructing feature models. The purpose of our proposed approach is to automate this process as much as possible.

[1] presents an industrial survey of dependencies between requirements for supporting release planning. Another two studies [6, 13] classify the relationships among features, which are used for analysis of dynamic dependencies between features. While in our approach, we classify the relationships between requirements for quantification of each relationship.

Clustering is a mature research subject itself [5, 18], which has been employed in many disciplines, including software engineering. It has been successfully used in reverse engineering [19] and objects/components identification [22, 14]. However, these methods focus on the design or implementation levels. Hisa and colleagues [9, 10] present a work closer to the requirements level. In this work, requirements are clustered into different groups, each of which is implemented as a module. This work uses clustering techniques for system decomposition and incremental delivery, while our approach uses clustering for feature identification and organization.

7. Conclusion

In this paper, we have presented a semi-automatic approach to constructing feature models based on requirements clustering, which is easy to understand and implement. In this approach, some sample applications are selected first. For each sample application tight-related individual functional requirements are clustered into features, and functional features are organized into an application feature model. Then all the application feature models are merged into a domain feature model, and the variable features are also labeled. The originality of our approach rests on that it exploits automated techniques including clustering techniques in the process of feature model construction to automate the activities of feature identification, organization and variability modeling to a great extent.

This approach has been applied to the data-dominant domains, and has been evaluated from two aspects. From the experimental results, we find that although the construction inevitably involves human intervention, our proposed approach can help construct feature models more effectively and generate good-quality features (clusters).

The proposed approach is a work in progress, and future effort is needed. First of all, we hope to integrate this approach into our feature-oriented method [16]. Additionally, because quality features often crosscut several functional features and are relatively complex, we haven't handled quality features in this approach. We plan to extend this approach to model them in the future. Finally, we need to apply this approach in more software domains. And in the further practice, the approach may need to be refined to be applicable to different domains. For example, as we study more software domains, we may find out more relationships between requirements and incorporate these relationships into our approach.

Acknowledgements

This work is sponsored by the National Basic Research Program (973) of China under Grant No. 2002CB312003; the National Natural Science Foundation of China under Grant No. 60233010, 60125206, 90412011; the Natural Science Foundation of Beijing under Grant No. 4052018.

References

[1] P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, and J. N. och Dag. An Industrial Survey on Requirements Interdependencies in Software Product Release Planning. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, pages 84–91, Toronto, Canada, August 2001. IEEE Computer Society Press.

[2] G. Chastek, P. Donohoe, K. C. Kang, and S. Thiel. Product Line Analysis: A Practical Introduction. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, 2001.

[3] M. G. Christel and K. C. Kang. Issues in Requirements Elicitation. Technical Report CMU/SEI-92-TR-12, Software Engineering Institute, Carnegie Mellon University, 1992.

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

[5] B. S. Everitt. *Cluster Analysis*. Edward Arnold, London, 3rd edition, 1993.

[6] S. Ferber, J. Haag, and J. Savolainen. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In *Proceedings of the Second Software Product Line Conference (SPLC 2)*, volume 2379 of *Lecture Notes in Computer Science*, pages 235–256, San Diego, CA, USA, August 2002. Springer-Verlag.

[7] M. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, Victoria, BC, Canada, 1998. IEEE Computer Society Press.

[8] A. P. Gupta and P. C. Grabow. Onion: A Methodology for Developing Data-Dominant Systems from Building Blocks. In *Proceedings of the conference on TRI-Ada '94*, pages 361–372, Baltimore, Maryland, USA, November 1994. ACM Press.

[9] P. Hisa and A. T. Yaung. Another Approach to System Decomposition: Requirements Clustering. In *Proceedings of the 12th Annual International Computer Software and Applications Conference (COMPSAC'88)*, pages 75–82, Chicago, IL, USA, 1988.

[10] P. Hsia and A. Gupta. Incremental Delivery Using Abstract Data Types and Requirements Clustering. In *Proceedings of the Second International Conference on Systems Integration*, pages 137–150, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[11] 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.

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

[13] K. Lee and K. C. Kang. Feature Dependency Analysis for Product Line Component Design. In *Proceedings of the 8th International Conference on Software Reuse (ICSR 2004)*, volume 3107 of *Lecture Notes in Computer Science*, pages 69–85, Madrid, Spain, July 2004. Springer-Verlag.

[14] J. Luo, R. Jiang, L. Zhang, H. Mei, and J. Sun. An experimental study of two graph analysis based component capture methods for object-oriented systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 390–398, Chicago, IL, USA, September 2004. IEEE Computer Society Press.

[15] A. Mehta and G. T. Heineman. Evolving Legacy System Features into Fine-Grained Components. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 417–427, Orlando, Florida, USA, May 2002. ACM Press.

[16] 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)*, pages 250–255, Dallas, Texas, USA, November 2003. IEEE Computer Society Press.

[17] H. Mili, F. Mili, and A. Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[18] B. Mirkin. *Mathematical Classification and Clustering*. Kluwer Academic Publishers, 1996.

[19] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A Reverse-engineering Approach to Subsystem Structure Identification. *Software Maintenance: Research and Practice*, 5:181–204, 1993.

[20] Object Management Group, Unified Modeling Language Specification, Version 1.5, 2003. Available at http://www.omg.org/technology/documents/formal/uml.htm.

[21] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A Conceptual Basis for Feature Engineering. *Journal of Systems and Software*, 49(1):3–15, December 1999.

[22] A. van Deursen and T. Kuipers. Identifying object using cluster and concept analysis. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 246–255, Los Angeles, CA, USA, May 1999. ACM Press.

[23] L. Wen and R. G. Dromey. From Requirements Change to Design Change: A Formal Path. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 104–113, Beijing, China, September 2004. IEEE Computer Society Press.

[24] K. E. Wiegers. *Software Requirements*. Microsoft Press, 1999.

[25] 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)*, volume 3308 of *Lecture Notes in Computer Science*, pages 115–130, Seattle, WA, USA, November 2004. Springer-Verlag.