A Component Retrieval Method based on Feature Tree Matching

Zhong-Jie Wang, De-Chen Zhan, and Xiao-Fei Xu

Research Centre of Intelligent Computing for Enterprises (ICE), School of Computer Science and Technology, Harbin Institute of Technology
P.O.Box 315, No. 92 West Da Zhi Street, Harbin, P.R.China, 150001

{rainy, dechen, xiaofei}@hit.edu.cn

Abstract

Component-related techniques have been considered as the mainstream of software reuse. In this paper, our main concern is to develop a component retrieval method to obtain a set of fine-grained components aiming at a coarse-grained requirement. Feature space is adopted to uniformly describe semantics of components and business requirements, therefore the problem of component retrieval is transformed to the problem of feature tree matching. There are seven steps in our method, in which a greedy-based approximation algorithm is used to realize performance optimization, i.e., minimizing matching cost for the final selected components, under the guarantee of maximizing function satisfaction. This method eliminates some deficiencies in traditional component retrieval methods, e.g., aiming at atomic function matching only, and ignorance on performance considerations.

Keywords: Component retrieval, Feature tree matching, Matching cost, Reuse

I. Introduction

As the component repositories scaling up and the reuse practice deepening, querying and retrieving components with high efficiency and accuracy to satisfy specific business requirement for short software delivery time and high software quality gains more attentions from software engineering researchers, and there have been a large quantity of component retrieval algorithms in literatures, such as behavior sampling based retrieval [1], signature and specification matching based retrieval [2], etc. Many related techniques, e.g., neutral network [3], fuzzy mathematics [4], reformulation and spreading activation [5], etc., have been adopted in component retrieval, too. Currently the most widely used method is faceted classification scheme based algorithms, with two typical strategies, i.e., (1) realizing relaxed matching based on traditional database query techniques combining synonyms dictionary and hierarchy structure of term space [6][7]; (2) transforming the matching between component specification and requirements into matching problem between faceted tree and using tree matching techniques [8][9] to realize component retrieval [10].

By practical application and analysis, we consider that there are the following two deficiencies in most of these component retrieval algorithms:

(1) There is an underlying hypothesis in all these methods that, the granularity of requirement models are in the same level as the granularity of components in libraries, and cannot be decomposed further, i.e., an atomic function. During algorithm execution, the faceted information of this atomic function is matched with each component in repository, and a single component (or, a set of candidate components with similar functions but different implementations) for the atomic requirement is retrieved.

However, this hypothesis is not true, i.e., in practice the granularity of a requirement model is usually far higher than components, and it cannot be realized by a single component but actually the composition of several components. When this situation appears, the requirement model has to be decomposed manually, and then a retrieval algorithm is adopted to query a proper component for each sub-model one by one. This leads to poor query efficiency.

(2) These methods primarily pay most of their attentions to function satisfaction with the metrics Precision and Recall, while ignore the performance optimization, i.e., in the situation that several components could satisfy functional requirements with the same satisfaction, these methods seldom consider the problem of how to specify which of them is finally chosen with the optimal performance during its reuse. Additionally, because each time only one part of requirement model is dealt with isolatedly and relationships between different parts are ignored, each the final selected component is only the local but not global optimization.

To solve these deficiencies, we present a feature tree matching based component retrieval algorithm (FTM), in which we emphatically consider the situation that the granularity of requirement model is much coarser than components, and transform component retrieval into the matching between features of requirement model and components. Function matching is first taken into account by obtaining a set of candidate components using tree matching; later, in order to evaluate the matching cost to realize performance optimization, some metrics for matching cost are adopted, according to which, the optimal components with low matching cost are finally picked out from candidate ones. In addition, we address the method of how to generate modification operation lists for the final selected components automatically and completely when these components cannot fully support the requirement model.

Rest of this paper is organized as follows. In section II a unified feature-oriented component model with its reuse mechanism and reuse styles are put forward as the foundation of our discussion. In section III, the 7-phase feature tree matching based component retrieval method with its key techniques are fully addressed. Section IV addresses a practical case for performance comparisons between our methods and traditional ones. Finally is the conclusion.

II. Feature-Oriented Business Component Model and its reuse styles

A. A unified feature-oriented business component model

Currently there exist various component semantics models, e.g., 3C [11], Wright [12], JBCOM [13], etc, each of which has their emphasis on describing business semantics contained in components. In order for succinct and clear explanations, we simplify and unify these models to get a *unified feature-oriented component model*, i.e., no matter what kind of heterogeneous component models, they can be uniformly expressed as the form of feature space, and most of characteristics of component models can be realized by utilizing advantages of feature modelling [14], such as:

(1) Functions implemented by components can be uniformly expressed as features *f*, and all features contained in a component *C* form the feather space $\Omega(C)$ of *C*.

(2) Composition relationships between functions. $\Omega(C)$ could be represented as a feature tree, in which "a function uses another function" can be expressed as "*Parent-Child*" features, i.e., if *f* uses *g*, then $g \in \text{child}(f)$, $f \in \text{parent}(g)$. Similarly there are "*Ancestor-Descendant*" and "*Sibling*" relationships between features, i.e.,

$$ancestor(f) = \begin{cases} \bigcup_{g \in parent(f)} ancestor(g), & parent(f) \neq \varnothing \\ \varnothing, & parent(f) = \varnothing \end{cases},$$

$$descendant(f) = \begin{cases} \bigcup_{g \in child(f)} descendan(g), & child(f) \neq \varnothing \\ \varnothing, & child(f) = \varnothing \end{cases},$$

$$sibling(f) = \{g | parent(g) = parent(f)\}.$$

(3) Reuse mechanism. A feature *f* may contain multiple feature items, denoted as $\text{dom}(f)=\{\tau_1,$

$\tau_2,\ldots, \tau_n\}$, each of which is a variable implementation of f. If f has only one feature item, i.e., $|\text{dom}(f)|=1$, f is called a fixed feature, or the commonality of the component; if f has multiple feature items, i.e., $|\text{dom}(f)|>1$, f is called a variable features, or the variability of the component. We have VARIATION_PART(C)=$\{f \mid |\text{dom}(f)|>1\}$ and FIXED_PART(C)= $\{f \mid |\text{dom}(f)|=1\}$.

(4) Dependencies between component functions could be expressed as the forms of feature dependency (FD), i.e., $X \rightarrow Y$ means that all features in Y are dependent on features in X.

(5) Component interfaces. A component may have two types of interfaces: PROVIDING and REQUIRED interfaces. A component provides its features to other components or environment via its PROVIDING interfaces and accesses features of other components via REQUIRED interfaces.

Therefore, a component can be denoted as the form of C=<f_{root}, F, FD, PS, RS>, just as illustrated in Figure I, where f_{root}, F, FD, PS, RS are the foot feature of C, feature set, feature dependency set, PROVIDING interfaces and REQUIRED interfaces, respectively. Figure I shows a simple example.

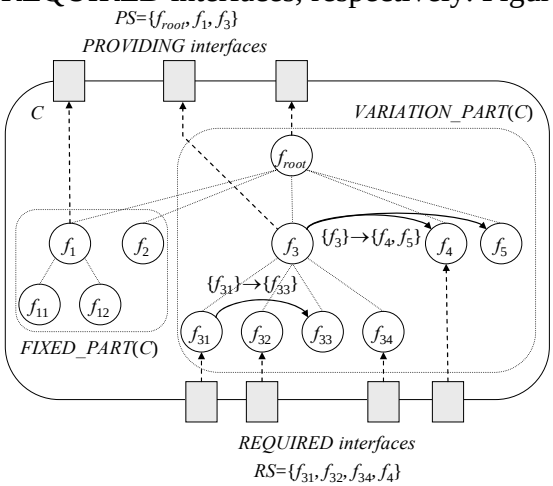


Figure I. An example of feature-oriented component

A reusable component is actually an abstract software artifact, which cannot be reused directly, and only after each variable features in C has been instantiated as a specific feature item, can C be reused. Therefore, the final reused component is in fact an instance of this component. Denote all the instances of C as instance(C)=$\{t_1, t_2,\ldots, t_p\}$. A component instance could be considered as a set consisted of one feature item of each feature in C.

B. ***Classification of component reuse styles***

Component reuse styles can be classified into the following four types, also called four *modification levels* of component reuse, according to the consistency degree between functions that a component could provide and the practical requirements:

(1) *Directly Reuse*: directly reuse components to construct new requirements without any instantiation or modification on these components.

(2) *Reuse after instantiation*: reuse an instance of an abstract component after instantiating it, i.e., choosing one specific feature item for each variable feature.

(3) *Reuse after modification*: an existing component cannot fully satisfy the requirements, so some modifications must be done before the component is really reused. The concrete modification means include:

- *Add new child features;*
- *Modify child features;*
- *Add new feature items;*
- *Modify feature items;*
- *Add new FDs;*
- *Delete FDs;*
- *Modify FDs.*

(4) *No reuse*: all existing components cannot completely satisfy requirements, therefore there are no reuses and new components must be designed for the requirement.

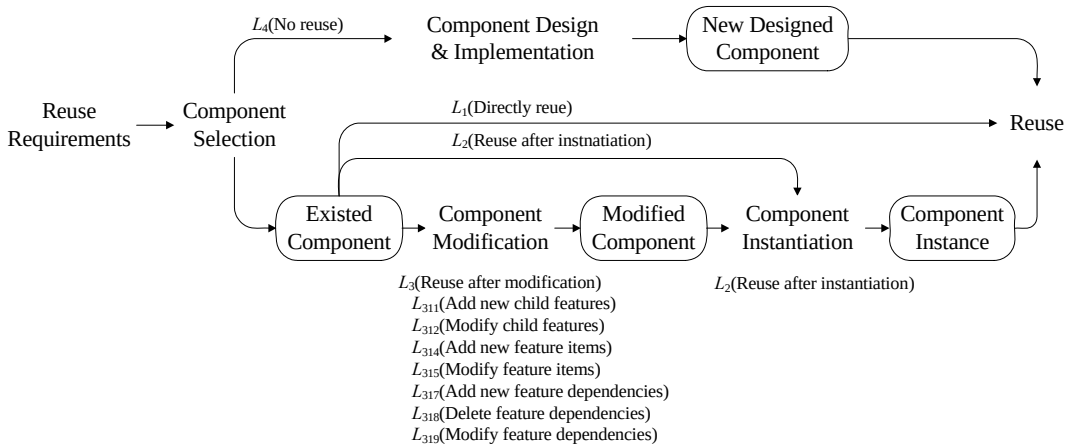


Figure II. Primary process and styles for component reuse

In conclusion, the process of component reuse can be divided into four phases: *component selection*, *modification*, *instantiation*, and *reuse*, just as shown in Figure II.

III. Feature Tree Matching based Component Retrieval Method

Problem Description: Specify the feature space $\Omega(bm)$ of a requirement business model *bm*, then pick a set of components $\{C_{p1}, C_{p2}, \ldots, C_{pm}\}$ from repository to ensure that $\{C_{p1}, C_{p2}, \ldots, C_{pm}\}$ could realize *bm* by modification and composition, i.e., $\Omega(bm) \subseteq \bigcup_{i=1}^{m} \Omega(C_{pi})$, with the optimal performance.

Easily to know, the final retrieved components may satisfy one of the following cases:

Case 1: $\Omega(bm) = \bigcup_{i=1}^{m} \Omega(C_{pi}), m > 1$, denoting that *bm* should be implemented by composition of multiple components;

Case 2: $\Omega(bm) = \bigcup_{i=1}^{m} \Omega(C_{pi}), m = 1$, denoting that *bm* may be exactly implemented by a single component;

Case 3: $\Omega(bm) \subset \bigcup_{i=1}^{m} \Omega(C_{pi}), m = 1$, denoting that *bm* can be implemented by part of a component.

In most situations, because the average granularity of business models is far coarser than that of components, therefore the first case appears with the largest probability, and the last case seldom appears. In the following, we emphatically consider Case 1 and 2, and ignore Case 3.

The performance of component selection is reflected by the following principles:

- Maximize function satisfaction degree: the *basic goal*;
- Minimize modifications of components: to decrease *modification level*/*cost*;
- Minimize number of final picked components, i.e., *components with coarse granularity first*: to improve *composition efficiency*;
- Minimize number of interactions between picked components: to decrease *composition cost*;
- Minimize redundancy: to decrease the scale of the final system.

By comparisons between these metrics, optimal component can be picked from multiple similar candidate ones.

The basic idea of component retrieval method in this paper is presented as follows. Firstly, according to the requirement, construct feature space of requirement model referring to the glossary of domain features; then pick each component from repository one by one and use the matching algorithm to specify whether this component can be used for the requirement model; if yes, then add it into candidate component set and calculate its matching cost, by which all candidate components are reordered form low to high. Repeatedly choose the first one in the ordered candidate component sequence to fill in the requirement feature space and again calculate the matching cost for the remainder candidate components relative to the remainder requirement feature space, until no candidate components are left or the requirement feature space has been fulfilled completely, and those components used in each filling phase are the final components. For each of picked components, produce its modification operation list by comparing with the requirement model. If the

requirement feature space has not yet been filled, new components should be designed. All the final components (selected from repository plus new designed) are composed together to implement the final system.

The process is shown in Figure III, and in next several sessions we will introduce key concepts and algorithms in each phase.

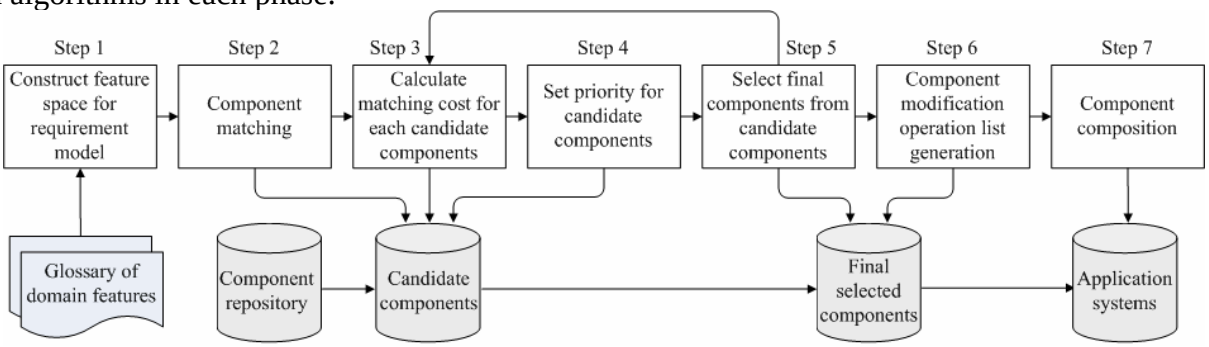


Figure III. Process of component retrieval for a coarse-grained requirement

A. Matching between components and requirement model

As mentioned above, a business component is represented as the form of feature space. Actually, a business model can also be represented as the same form [15], and a component may be considered as a sub-space of a business model. Therefore, the problem of choosing components from repository to construct requirement model, can be transformed into the problem of matching between feature space of components and requirement model, i.e., judging whether a specific component can be used as a part of the requirement model. Because feature space may be denoted as the form of feature tree, therefore the matching process is considered as the matching between two trees [16]. Here we introduce some basic concepts.

Definition 1 (*Common Ancestor Features*). If there does not exist *ancestor-descendant* relationship between f_1 and f_2, then use A(f_1, f_2) as their common ancestor features, and $A(f_1, f_2) = ancestor(f_1) \cap ancestor(f_2)$.

Definition 2 (*Relative Closeness*). If there does not exist *ancestor-descendant* relationship between f_1 and f_2, then use R(f_1, f_2) as the relative closeness between them, and $R(f_1, f_2) = |A(f_1, f_2)|$. The more common ancestor features they have, the closer the relationship between them is.

In the following discussions, suppose T is the feature tree of requirement model *bm*, T_{sub} is a sub-tree of T, and F_C is the feature set of component C. There are five types of tree matching between T and F_C [10], and we extend and present them below:

Definition 3 (*Embedded Matching*, EM). If there is a mapping p from F_C to T_{sub} satisfying the following constraints, then p is called an embedded matching from F_C to T:

(1) $\forall f_1, f_2 \in F_C, f_1 = f_2 \Leftrightarrow p(f_1) = p(f_2)$

(2) $f = p(f)$

(3) $f_1 = parent(f_2) \Leftrightarrow p(f_1) = parent(p(f_2))$

(4) $|child(f)| = |child(p(f))|$

(5) $dom(f) \supseteq dom(p(f))$

(6) $FD_T \subseteq FD_C$

Embedded matching is an isomorphism from the feature space of a component to a subspace of the requirement model, i.e., this component may fully realize part functions of the requirement model.

Definition 4 (*Area Matching*, AM). If there is a mapping p from F_C to T_{sub} satisfying the following constraints, then p is called an area matching from F_C to T:

(1) $\forall f_1, f_2 \in F_C, f_1 = f_2 \Leftrightarrow p(f_1) = p(f_2)$

(2) $f = p(f)$

(3) $f_1 = parent(f_2) \Leftrightarrow p(f_1) = parent(p(f_2))$

(4) $dom(f) \supseteq dom(p(f))$

(5) $FD_T \subseteq FD_C$

Based on embedded matching, area matching relaxes constrains on mapping, i.e., the component may realize part functions of a sub-tree in the requirement model.

Definition 5 (*Containment Matching*, CM) If there is a mapping p from F_C to T_{sub} satisfying the following constraints, then p is called a containment matching from F_C to T:

(1) $\forall f_1, f_2 \in F_C, f_1 = f_2 \Leftrightarrow p(f_1) = p(f_2)$

(2) $f = p(f)$

(3) $f_1 = ancestor(f_2) \Leftrightarrow p(f_1) = ancestor(p(f_2))$

(4) $dom(f) \supseteq dom(p(f))$

(5) $FD_T \subseteq FD_C$

Definition 6 (*Strong Constrained Containment Matching*, SCCM) If p is a containment matching from F_C to T, and satisfies that if there are no *ancestor-descendant* relationships between f_1, f_2 and f_3, and $R(f_1, f_2) = R(f_1, f_3) \Leftrightarrow R(p(f_1), p(f_2)) = R(p(f_1), p(f_3))$, then p is called a strong constrained containment matching from F_C to T.

Definition 7 (*Weak Constrained Containment Matching*, WCCM) If p is a containment matching from F_C to T, and satisfies that if there are no ancestor-descendant relationships between f_1, f_2 and f_3, and $R(f_1, f_2) < R(f_1, f_3) \Leftrightarrow R(p(f_1), p(f_2)) \leq R(p(f_1), p(f_3))$, then p is called a weak constrained containment matching from F_C to T.

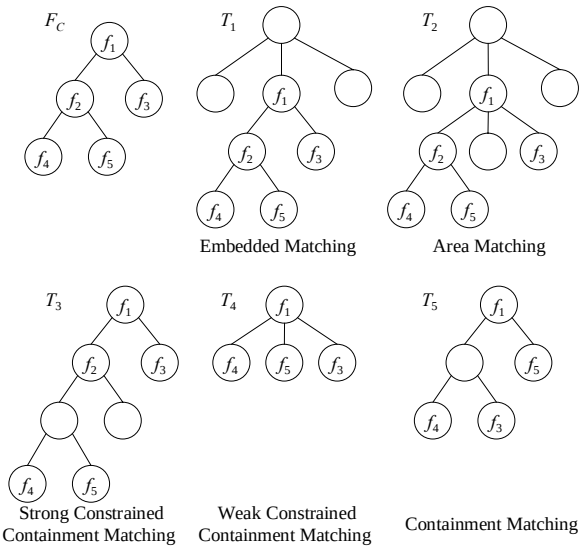


Figure IV. Examples of five types of tree matching [10]

Relationships between above five types of matching are EM→AM→SCCM→WCCM→CM, where the arrow points from closer to more relaxed matching. Constraints on these mapping are gradually relaxed from strict parent-child relationship (e.g., EM and AM) to ancestor-descendant (e.g., SCCM, WCCM and CM) in vertical, and from the constraints on strict child node number equality (e.g., EM and AM) relaxed to constraints on relative closeness equality (e.g., SCCM) until relaxed to none (e.g., CM) in horizon. Comparisons between these five types of matching are: the final chosen components' modification degree is more and more great, but the number of new designed components are gradually decreased. In practice, we may adopt different mapping strategies according to different emphasis.

Some examples of these mappings are shown in Figure IV.

Algorithm 1: Candidate components generation, $CCG(T, \{C_1, C_2, \ldots, C_n\}, \{F_{C_1}, F_{C_2}, ..., F_{C_n}\})$.

Input: Feature tree T of requirement model, a set of components $\{C_1, C_2, \ldots, C_n\}$ contained in repository and the corresponding feature tree $\{F_{C_1}, F_{C_2}, ..., F_{C_n}\}$ of each component.

Output: a set of candidate components CC_T and the matching cost $\gamma(C_i,T)$ for each candidate component.

```
do
{
    Choose one component C_i from {C_1,C_2,…,C_n};
    Search the root feature f of C_i in T;
    if(no found)
            break;
    else {
            Suppose the found feature is p(f), then construct a sub tree T_sub of T containing p(f) and its all
            descendant features descendant(p(f)).
            Judge whether there exists a mapping from F_Ci to T_sub according to Definition 3~7;
            if(no)
                    continue;
            else {
                    Calculate the matching cost γ(C_i,T) using Algorithm 2 and 3;
                    Add C_i into CC_T;
            }
      }
}while(all the components in {C_1,C_2,…,C_n} has been dealt with)
```

In this algorithm, aiming at each component, we firstly look for its root feature f in the requirement model, and if there does not exist, it shows that this component cannot be used to construct the model at all. If it does, then try to construct mapping between the component and the sub-tree with the root node f. If there exists a mapping (one of the five types), then calculate the matching cost using Algorithm 2 and 3 (presented in next two sections) and add it into candidate component set.

B. Matching cost for candidate components

All candidate components obtained by Algorithm 1 have function satisfactions with requirement models. However, component selection requires not only function matching, but also the optimal performance. Therefore, it is necessary to calculate the cost of each candidate component for constructing requirement model. The matching cost $\gamma(C,T)$ is not a single value, but composed of a set of metrics, i.e.,

- *Matching ratio* $mr(C,T)$, the ratio that the number of features useful to T and contained in C relative to the number of features contained in the whole T, denoted as $mr(C,T)=\left|\Omega(C)\cap\Omega(bm)\right| \Big/ \left|\Omega(bm)\right|$, $mr(C,T)\in(0,1]$. A larger $mr(C,T)$ means that C has greater contributions to the construction of T, and when multiple components all could fulfill one part of requirement model, those components with coarser granularity is sure to have higher matching ratio than finer ones;
- *Feature redundancy* $rud(C,T)$, the ratio that the number of features contained in C but useless to T relative to the number of useful features in C, denoted as $rud(C,T)=\left|\Omega(C)-\Omega(bm)\right| \Big/ \left|\Omega(C)\cap\Omega(bm)\right|$, $rud(C,T)\in[0,+\infty)$. A larger $rud(C,T)$ means that C contains much more redundant and useless features to T;
- *Interface dependency* $itd(C,T)$, the number of features C requires from other components via its REQUIRED interfaces but useless to T, denoted as $itd(C,T)=\left|RS(C)-\Omega(bm)\right|$. A larger $itd(C,T)$ means that when C is used to construct T, some other components containing features that C requires but are useless to T have to be selected along with C. This forces us to query these components again from repository or implement them manually, leading to extra works.
- *Coupling* $cou(C,T)$, the number of required or providing features contained in C and useful to T, denoted as $cou(C,T)=\left|RS(C)\cap\Omega(bm)\right|+\left|PS(C)\cap\Omega(bm)\right|$. A larger $cou(C,T)$ means that when C is used to construct T, some coupling connections must be created between C and other selected components

via their PROVIDING or REQUIRED interfaces, therefore leads to higher composition cost.

- *Modification cost moc*(*C,T*). Generally speaking, the possibility of obtaining components that are fully consistency with requirements is comparatively small. *moc*(*C,T*) refers to the cost to modify or adjust *C* to precisely satisfy requirements in *T*. *moc*(*C,T*) can be divided into six parts:

(1) *fdc*(*C,T*): cost for deleting those redundant features in *C*;

(2) *aic*(*C,T*): cost for adding new feature items which is required in *T* but not yet contained in *C*;

(3) *mic*(*C,T*): cost for modifying those feature items contained in *C* but not fully support the required feature items in *T*;

(4) *adc*(*C,T*): cost for adding new feature dependencies;

(5) *mdc*(*C,T*): cost for modifying feature dependencies;

(6) *ddc*(*C,T*): cost for deleting feature dependencies.

Therefore, $moc(C,T)=fdc(C,T)+aic(C,T)+mic(C,T)+adc(C,T)+mdc(C,T)+ddc(C,T)$

Algorithm 2: Matching cost calculation, *MCC*(*T, C*)

Input: the requirement model *T* and a candidate component *C*

Output: the value of each metrics for matching cost, i.e., *mr*(*C,T*), *rud*(*C,T*), *itd*(*C,T*), *cou*(*C,T*) and *moc*(*C,T*).

Use those formulas in above definitions to get the results.

C. Priority setting for candidate components

A component is considered as candidate component only indicates that this component may be used in *T*'s construction, and in reality there usually exists such a situation that, several similar components all could realize a specific function, then which one of them is the best? In this section we synthesize all the metrics presented in last section, according to which to order candidate components, and a candidate component near to the beginning of the sequence is considered to have higher priority than those ones near to the tail.

We consider that the five metrics of matching cost satisfies the following relationships: $mr(C,T) >> moc(C,T) > \frac{rud(C,T)}{itd(C,T)} > cou(C,T)$, i.e., when we select components, we should first consider the degree that a component contributes to the requirement model, i.e., matching ratio; then modification cost, i.e., those selected components may participate in the construction process without large scale modifications; then consider feature redundancy and interface dependency to give up those candidate components with high redundancy and high dependencies on unrelated components; finally is the coupling.

Under the guidance of this principle, we set the corresponding weight for each metrics, i.e., $\beta_{mr}=10$, $\beta_{moc}=4$, $\beta_{rud}=\beta_{itd}=2$, $\beta_{cou}=1$, therefore, the integrated matching cost may be calculated by the following equation:

$$\gamma(C,T)=\frac{\beta_1\times\left(1-\exp\left(-\frac{1}{M_1(C,T)}\right)\right)+\sum_{i=2}^{5}\left(\beta_i\times\left(1-\exp\left(-M_i(C,T)\right)\right)\right)}{\sum_{i=1}^{5}\beta_i}$$

where $M_i(C,T)$ represents each of the five metrics, with the corresponding weight β_i respectively. 1-exp(-*x*) is used to normalize the value *x* into the domain [0,1].

Algorithm 3: Calculating priority of each candidate component, *CPCC*(CC_T)

Input: a set of candidate components CC_T and each one's metrics for matching cost *mr*(*C,T*), *rud*(*C,T*), *itd*(*C,T*), *cou*(*C,T*) and *moc*(*C,T*)

Output: candidate component sequence $OrderedCC_T$ ordered by matching cost from low to high

Step 1: Using above formula to calculate the integrated matching cost $\gamma(C,T)$ for each component *C* in CC_T;

Step 2: Order CC_T according to $\gamma(C,T)$ and output to $OrderedCC_T$ from low to high order.

D. ***Final component selection from candidate component set***

After obtaining the ordered candidate components, the next task is to determine which components are finally chosen for the construction of requirement model. Because the problem of tree matching has been proved as an *NP-complete* problem [8], we adopt an *approximation algorithm* based on *greedy* principle, i.e., in each iteration we pick the component with lowest matching cost out and re-calculate the matching cost for the left candidate components, until all the candidate components have been dealt with (chosen or discarded) or the requirement model has been completely fulfilled.

Algorithm 4: Final component selection, $FCS(T, OrderedCC_T)$

Input: the requirement model T, the ordered candidate component sequence $OrderedCC_T$.

Output: the final chosen components FC_T and those unsupported features T' in T by FC_T.

```
Let FC_T=∅, T'=∅;
while(1) {
    if(OrderedCC_T=∅ or T=∅){
        T'=T;
        exit;
    }
    if(OrderedCC_T≠∅ and T≠∅){
        Choose the first component C in OrderedCC_T;
        Remove those intersected features (between T and C) from T, i.e., let T=T−T∩Ω(C);
        ∀f∈T∩Ω(C), let flag(f)='U' which indicates that f is a useful feature for T;
        Delete C from OrderedCC_T and add it to FC_T, i.e., OrderedCC_T=OrderedCC_T−{C}, FC_T=FC_T∪{C};
    }
    Call Algorithm 2 and Algorithm 3 to re-calculate and re-order the matching cost of the remainder candidate
    components in OrderedCC_T;
}
```

E. ***Modification operation list generation***

In Algorithm 4 we have specified FC_T, i.e., the final selected candidate components, but these components cannot be reused directly, and the difference between these components and requirement model demands us to modify these components by a series of operations. In addition, from Step 2 in Algorithm 4 we can know that if the algorithm exists when $OrderedCC_T=\varnothing$, it shows that there are still some features (i.e., features contained in T' when the algorithm exits) in T that are not yet realized by components in FC_T, therefore we have to design and implement new components for these features. In this section we present the algorithm for obtaining the modification operation list of FC_T.

Algorithm 5: Modification operation list generation, $MOLG(T, FC_T, T')$

Input: the requirement model T, the final component set FC_T and unrealized feature set T'

Output: modification operation list $ModOP_T$

Step 1. Order features in T' according to levels from high to low in T's feature space.

Step 2. Select the first unmatched feature f from T' and create modification operations according to the following rules:

Rule 1. If $child(f)=\varnothing$ and $parent(f)\notin T'$, there must $\exists C\in FC_T$ which makes $parent(f)\in\Omega(C)$, then let $\Omega(C)=\Omega(C)\cup\{f\}$ and create MOP $modOP(AS,\{f\},C)$;

Rule 2. If $child(f)\neq\varnothing$, then create a new component C and let $\Omega(C)=UnD(f)$, $UnD(f)=\{f\}\cup\{g|g\in descendant(f)\cap T'\wedge parent(g)\in UnD(f)\}$, with an MOP $modOP(AC,UnD(f),C)$;

Rule 3. If $child(f)\neq\varnothing$ and $child(f)\cap T'=\varnothing$, $parent(f)\cap T'=\varnothing$, then f is discarded.

In *Rule* 1~3 above, f's domain (i.e., its feature items) is set to contain the corresponding feature item in requirement model T.

Step 3. Delete f from T', add all the new designed components into FC_T, and add all the operations into $ModOP_T$;

Step 4. Execute Step 2 and 3 recursively until $T'=\varnothing$. Now all the features in T have been matched;

Step 5. Take one feature f from T (suppose that f has the required feature item τ and f is supported by component C in FC_T), and check whether $\tau \in dom(f)$ is true in C. If $\tau \notin dom(f)$ but there exist $\tau' \in dom(f)$ and τ may be obtained by modifying τ', then add MOP $modOP(MI, \tau, \tau', f, C)$ in $ModOP_T$, denoting to modify f's feature item τ' to τ. If there does not exist such a τ', then add MOP $modOP(AI, \tau, f, C)$ to $ModOP_T$, denoting to add a new feature item τ for f in C. Repeat this step until all the features in T have been checked.

Step 6. Take one feature dependency fd from T (suppose fd has the form $X \rightarrow Y$) and check whether there exists the corresponding feature dependency in feature spaces of components in FC_T. Similar with the strategies in Step 5, if there does not exist, then add $modOP(AD, fd, X, Y)$ to $ModOP_T$, denoting to add a new feature dependency fd between feature sets X and Y in components FC_T; if there is a similar feature dependency fd', then add $modOP(AD, fd, fd', X, Y)$ to $ModOP_T$, denoting to modify fd' to fd. Repeat this step until all the feature dependencies in T have been checked.

Step 7. Check each feature dependency fd contained in components in FC_T to see whether fd appears in T. If no, then add the MOP $modOP(DD, fd, X, Y)$ in $ModOP_T$, denoting to delete this useless feature dependency.

Step 8. The algorithm ends, and now $ModOP_T$ contains all the MOPs, i.e., constructing what new components and how to modify those selected components. Do the modifications on components in FC_T according to $ModOP_T$.

F. ***Component composition***

The last step is component composition, i.e., composing all the components in FC_T via interface connection to form the requirement model T.

Algorithm 6: Component Composition, $CC(T, FC_T)$

Input: requirement model T and the final chosen components FC_T

Output: the composition of these components

Step 1. For each component C in FC_T, check C's each REQUIRED interface, if there is a feature f that are still not yet satisfied from other components' PROVIDING interfaces, then use Algorithm 1~4 to choose a corresponding component C' which may provide f to C. Repeat executing this step until there are no such features and FC_T does not change any longer.

Step 2. Create interface connections between PROVIDING and REQUIRED interfaces of different components in FC_T.

IV. Comparisons with other methods

In this section we present a practical case. We adopted some segments of *Production Planning System* (PPS) in a component-based E*nterprise Resource Planning* (ERP) as requirement models, and use our feature tree matching based method (*FTM*) and *GenetricMatching* algorithm (*GM*, which is a faceted scheme based retrieval algorithm, we use it here as a representative of traditional retrieval methods in literatures) in [10] to retrieve components from our ERP component repository (containing 176 components in PPS domain).

In five test cases, the granularity of the requirement model, i.e., number of leaf features (denoted as n_f), grows gradually from 1, 2, 6, 11 to 19. We adopted *Strong Constraint Containment Matching* (SCCM) strategy in these tests. Figure V~VIII shows comparisons between *GM* and *FTM* on query times from component repository, total execution time, number of final selected components, and total matching cost of the final selected components, respectively.

From these figures we can draw the following conclusions:

(1) When the scale requirement model is growing coarser, *GM* has to be repeatedly executed once for each feature to be matched, therefore the query times equals to n_f and the execution time has linear relationship with n_f. For *FTM*, we only need to execute once to query a set of candidate components from repository no matter how larger n_f is, and most execution time is consumed by matching cost evaluation and final component selection from candidate set.

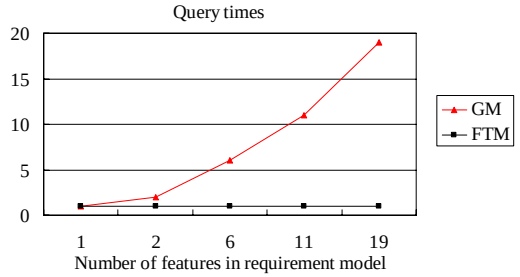


Figure V. Comparisons on query times

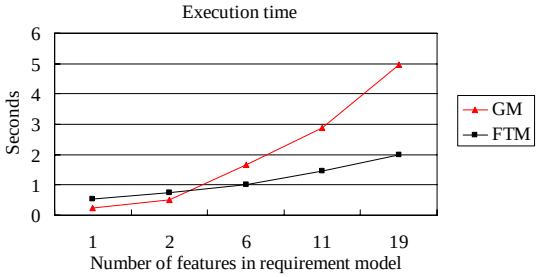


Figure VI. Comparisons on execution time

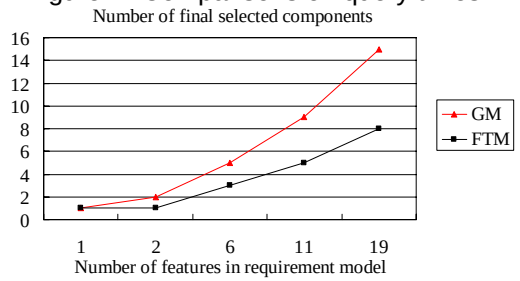


Figure VII. Comparisons on number of final selected components

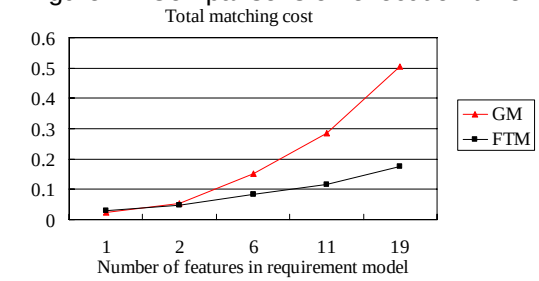


Figure VIII. Comparisons on matching cost

(2) When n_f is small (i.e., only 1 feature), the execution time of *FTM* is worse than *GM*. This is because in this situation, some factors, e.g., interface redundancy, feature redundancy, coupling, etc, are actually not required to be considered.

(3) *GM* did not consider the relationships between components when multiple components instead of one required to be retrieved from repository, therefore it cannot avoid high redundancy and coupling, which then lead to steeply increasing total matching cost with n_f. Contrarily, the matching cost is regarded as an important goal in our *FTM* method and is globally optimized.

(4) *GM* only retrieves components with the same granularity as atomic features, therefore, the number of final retrieved components is also linearly increasing with n_f; however in *FTM*, because it follows the principle of "components with coarser granularity have higher priorities", the number of selected components is quite smaller, which will lead to better composition efficiency.

V. Conclusions

Traditional component retrieval algorithms, e.g., faceted based methods, usually restrict the scale of requirement models into the level of reusable component, which cannot deal with coarser-grained requirements; in addition, these methods focus mainly on function satisfaction, but ignore optimization on some performance metrics, e.g., final components' modification degree, redundancy, composition cost, etc.

The feature tree based component retrieval methods presented in this paper eliminates these deficiencies. The whole process is divided into seven phase, in which by five types of matching between feature trees of requirement model and components, we firstly get a set of candidate components for the function satisfaction; then according to a set of performance metrics, matching cost for each candidate components is evaluated; later, a greedy-based approximation algorithm is adopted to pick out those candidate components with minimum matching cost iteratively.

Practical test cases show that, with the scale of the requirement model growing, our method has better performance than traditional methods in literatures.

Acknowledgement

Research works in this paper are partial supported by *the National Natural Science Foundation of China* (No. 60573086) and *the Specialized Research Fund for the Doctoral Program of Higher Education* (SRFDP) in China (No. 20030213027).

References

[1] Podgurski, A., Pierce, L., Retrieving reusable software by sampling behaviour, ACM Transactions on Software Engineering and Methodology, Vol.2, No.3, 2003, pp.286-303.

[2] Zaremski, A.M., Signature and specification matching, Ph.D Thesis, School of Computer Science, Carnegie Mellon University, 1996.

[3] Merkl, D., Tjoa, A.M., Kappel, G., Learning the semantic similarity of reusable software components, Proceedings of the 3rd International Conference on Software Reuse (ICSR'94), IEEE Computer Society Press, 1994, pp.33-41.

[4] Damiani, E., Fugini, M.G., Bellettini, C., A hierarchy-aware approach to faceted classification of object-oriented components, ACM Transactions on Software Engineering and Methodology, Vol.8, No.3, 1998, pp.215-262.

[5] Henninger, S., Supporting the process of satisfying information needs with reusable software libraries: an empirical study, Proceedings of the 17th International Conference on Software Engineering on Symposium on Software Reusability, ACM Press, 1995, pp.267-270.

[6] Prieto-Diaz, R., Implementing faceted classification for software reuse, Communications of the ACM, Vol.34, No.5, 1991, pp.88-97.

[7] Sorumgard, L.S., Sindre, G., Stokke, F., Experiences from application of a faceted classification scheme, Proceedings of the 2nd International Conference on Software Reuse, IEEE Computer Society Press, 1993, 24-26.

[8] Zhong, K.Z., Tao, J., Some MAX SNP-hard results concerning unordered labelled trees, Information Processing Letters, Vol.49, No.5, 1994, pp.249-254.

[9] Kilpelainen, P., Tree Matching problems with applications to structured text databases, Technical Report, A-1992-6, Department of Computer Science, University of Helsinki, 1992.

[10] Wang Y.F., Xue, Y.J., Zhang, Y., Zhu, S.Y., Qian L.Q., A matching model for software component classified in faceted schema, Journal of Software, Vol.14, No.3, 2003, pp.401-408.

[11] Tracz, W., Implementation Working Group Summary, Proceedings of Reuse in Practice Workshop, IDA Document D-754, 1994, pp.10-19.

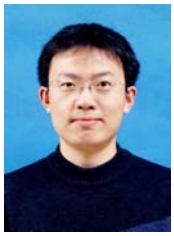
[12] Allen, R. and Garlan, D., A Formal Basis for Architectural Connection, ACM Transactions on Software Engineering and Methodology, Vol.6, No.3, 1997, pp.213-249.

[13] Wu, Q., Chang, J., Mei, H., Yang, F.Q., JBCDL: An Object-Oriented Component Description Language, Proceedings of 24th International Conference on Technology of Object-Oriented Languages ASIA, 1997, pp. 198-205.

[14] Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., FORM: A feature-oriented reuse method with domain-specific reference architectures, Annals of Software Engineering, Vol.5, 1998, pp.143-168.

[15] Wang, Z.J., Xu, X.F., Zhan, D.C., A Component Optimization Design Method based on Variation Point Decomposition, in Proceedings of the 3rd ACIS International Conference on Software Engineering, Research, Management and Applications, IEEE Computer Society Press, 2005, pp.399-406.

[16] Schlieder, T., ApproXQL: Design and Implementation of An Approximate Pattern Matching Language for XML, Technical Report, B01-02, Freie Universität Belin, 2001.



Zhong-Jie Wang is a lecturer in computer application technology in School of Computer Science and Technology at Harbin Institute of Technology (HIT), China. His research interests include software engineering, software reuse, software reconfiguration, software component related techniques.

De-Chen Zhan is a professor in School of Computer Science and Technology at Harbin Institute of Technology (HIT), China. His research interests include computer integrated manufacturing system (CIMS), enterprise resource planning (ERP), decision support systems (DSS), software reuse and reconfiguration, etc.

Xiao-Fei XU is a professor and dean of School of Computer Science and Technology at Harbin Institute of Technology (HIT), China. His research interests include computer integrated manufacturing system (CIMS), management and decision information system, software engineering, etc.