STCIM: A Dynamic Granularity Oriented and Stability Based Component Identification Method

Zhong-Jie Wang, De-Chen Zhan, Xiao-Fei Xu
School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
{rainy, dechen, xiaofei}@hit.edu.cn

Abstract

Among recent developments in the field of software reuse has been the increasing reuse of coarse-grained components, and it has been proved that granularity has great impact on component's reuse performance. However, previous studies have ignored rigorous and effective methods to support coarse-grained component identification and design, particularly granularity optimization design. In this paper, a stability-based component identification method, STCIM, is presented to resolve this problem. First a feature-oriented component model and the corresponding component granularity metrics are briefly presented. By establishing mappings between business model space and component space, component design process may be regarded as the process of decomposition, abstraction and composition of business model elements, with four different mapping strategies discussed to obtain dynamic component granularities. Furthermore, it is thought that component granularity is closely correlative to the stability of business models: the more stable the business model, the larger the corresponding component granularity may be. A metrics for model stability with three factors, i.e., number of isomers, stability entropy and isomer similarity, is presented, and the corresponding component identification algorithm based on *Most Stable Set* is discussed in details. Finally a practical case is described to validate the method in this paper.

Keywords: business component, component identification, feature modeling, stability, dynamic granularity

1 Introduction

With the rapid development of software reuse theories and techniques, the evolution of reusable software artifacts have ranged from functions, objects and classes in object-oriented programming, to components, frameworks, design patterns [1][2], services [3] and software architectures [4] in today's Internet-based environments. The granularity of reusable artifacts is continuously increasing [5]. Currently, coarse-grained reuse, such as component-based development [5][6] and service-oriented computing [7], has been the primary directions in software reuse [2].

Granularity depicts the scale and complexity of a reusable artifact and is difficult to be precisely measured. Qualitatively speaking, components may be classified into fine-grained, medium-grained and coarse-grained ones [8], called *granularity level*. For example, a process component provides a single, discrete business process, while an activity component provides one or more business activities, therefore the granularity of the former is higher than the latter. Different components in the same granularity level still have different granularities, which can be evaluated in two ways: the number of functions that a component can provide outward by its interfaces, and the number of reusable entities contained in the component. To some up, the coarser granularity a component has, the more functions it may provide, or the more entities it has implemented.

The reason that granularity is considered a key concept in software reuse is that it influences components' reuse performance to a great extent [1][6]. Components with finer granularity have wider range of reuse and higher reusability, however, to realize a large and complex system, numerous fine-grained components have to be time-consumingly composed together, therefore the reuse efficiency is lower and reuse cost is inefficient. In comparison, coarser granularity components certainly have higher reuse efficiency and lower reuse cost.

On the other hand, fine-grained components, such as business object components, depict those stable and core business elements in an enterprise and have experienced few changes during their whole lifecycle, therefore, when these components are reused, it is not required to be modified or reconfigured on a large scale. However, coarse-grained components, e.g., business activity components, business process components, contain a mass of unstable business elements such as business rules [9], which usually need to be changed regularly with rapidly changing business requirements. Therefore, when these components are reused, they have to be reconfigured or modified to a great extent, and will lead to higher reuse cost. Minimizing unstable elements contained in a component and maximizing component granularity will enable us to exploit its advantages and enhance its reusability at the same time.

Although coarse-grained reuse is the prevailing trend, coarser granularity does not necessarily equal better quality. Therefore, the proper specification of components to achieve maximum enhancement of their granularity while minimizing the limitations inherent in coarse-grained components is a critical challenge in the component identification and design process, which this paper is going to address.

Component identification is a primary mission in the phase of domain engineering, in which, domain business models are decomposed and clustered into a set of reusable components [10][11]. By reusing these components in the phase of application engineering, software systems that support specific business models can be implemented easily. Therefore, there is a strict and bi-directional mapping between business models and components, i.e., components are the software representation of business models, and business models are the semantics representation of component models. Due to the existence of this mapping, component granularity is closely related to the properties of business models.

This kind of mappings should satisfy *semantic consistency*. When components are reused to construct specific business models (or, software systems), if their semantics cannot reach consistency, i.e., the functions that a component provides cannot satisfy (or conflicts with) the requirements of the models, it becomes necessary to adjust inner structure and semantics of these components, thus causing higher reuse cost. From the statistical view, if a business model in a specific business domain needs frequent changes, then, those components obtained from this business model are sure to have a high probability of frequent changes, too. We refer to the probability that a business model or a component remains stable in a given period of time as "*stability*".

Component stability and business model stability are closely related. The higher stability a business model has the more unlikely those components obtained from the business model will change frequently, and vice versa. Therefore, it is better that "a component formed with highly stable business elements results in a coarser granularity, while frequent changing business elements will result in finer-grained components". Based on this idea, a component identification method, STCIM, is presented in this paper for granularity optimization based on business model stability. In this method, the stability of domain business models is used as the decision information in the process of component identification and granularity optimization. By calculating the stability of every business element and the degree of stability dependency between these elements, we map those stable elements into coarser-grained components, and for those instable elements, we iteratively decompose them into several stable clusters and map every one of them to a finer-grained component. The method could finally bring about multi-granularity component co-existence.

The rest of this paper is organized as follows. In section 2 related works in literatures are briefly introduced. In section 3, we present a feature-oriented business model and the corresponding definition of granularity based on feature space. In section 4, we discuss the mapping between business model space and component space, with four mapping strategies emphasized. In section 5 the definition of stability and stability dependency, and the corresponding metrics, are described in detail. A component design method based on stability, and a practical case, is finally presented in section 6. The conclusion is given in section 7.

2 Related works

2.1 Component Granularity

Researchers have proposed various methods to define and measure component granularity qualitatively and quantitatively. Generally speaking, component granularity is a metrics that is used to describe the degree of coarseness or thinness of those functions provided by a component [12].

The simplest evaluation way of component granularity is the length of interior business logic of the component, e.g., number of program lines [13]. In fact component granularity is difficult to be precisely measured. If a component is not self-contained, i.e., in order to use the component, some other components must also be reused at the same time, then it is a fine-grained component, otherwise it is a coarse-grained component [14].

In an approximate way, an entity component's granularity can be calculated by the number of business entities contained in this component. For a process component, it's granularity is increasing linearly with the number of branches and activities in the process diagram realized by the component [15], and can be calculated by the formula $\text{Granularity}(PC)=\sum_{u\in PC}[\alpha\times\text{Branch}(u)^2+\beta\times\text{Activity}(u)]$. Different types of components are in different granularity levels, and a coarse-grained component can be implemented by the composition of a set of fine-grained components. Therefore, granularity is a concept that should be recursive defined. Generally speaking, component granularity can be calculated by the synthesis of granularity of sub-elements contained in it.

2.2 Domain engineering and component identification method

Component identification and design is a primary mission in domain engineering (DE). Component designers carry out domain analysis from a set of similar requirements in a specific domain to find out the commonalities and variabilities in this domain and construct DSSA [16][17]. According to these results, reusable business semantics are obtained and the corresponding component specifications can be designed [2]. Some typical and widely applied DEmethods include FODA [18], FORM [19], ODM [20], RSEB [21], etc.

Actually the basic idea of DE is consistent with Model-Driven Architecture (MDA) [22]. Domain models depicts the common business requirements in a domain, and they belong to Computation-independent Model (CIM), while component-based software models may be considered as a kind of Platform-Independent Model (PIM), therefore, component identification can be regarded as the transformation from CIM to PIM in MDA. Research concerning this aspect is not extremely mature and complete.

It is not completely at will to cluster business models into components, and certain principles must be followed. The majority of current component design principles mostly come from design principles of class and class package in object-oriented methods, such as Open-Close Principle (OCP), Common Reuse Principle (CRP), Acyclic Dependency Principle (ADP), Single Responsibility Principle (SRP), etc [23]. But, a component is not a simple aggregation of classes; therefore these principles are not fully suitable for component design. In addition, these principles can be primarily used in the design of entity components, and is difficult to instruct the identification of process components. On the other hand, these principles are unable to accurately answer the question that what kind of granularity can lead to best reuse performance, and there also lack of strict and effective methods to support the identification and design of coarse-grained components.

The problem of component identification and design has aroused the widespread interest with the deep research on Component-based Software Engineering (CBSE) [2] in mid 1990s. Initially, component identification was regarded as a phase in domain engineering. These methods, e.g., FODA [18] /FORM [19], emphasize particularly on the reusability of DSSA and components' compatibility and adaptability between similar application families in a domain, but do not pay much attention to the optimization on other factors, such as reuse cost, reuse efficiency, etc, and granularity is not considered, neither.

In later researches, it has obtained full attentions as an independent problem. Starting from considering component reuse cost, researchers try to decompose business models into a set of sub-models according to those principles such as "high-cohesion and low-coupling" [6] by calculating the overall relevance between business elements, and encapsulate every sub-model into one component. Typical representatives include CRWD matrix based COMO [24] and O2BC [25] methods, and many clustering analysis based methods, e.g., minimal spanning tree techniques, graph clustering, and heuristic algorithms, etc, which are imported from other fields such as Artificial Intelligence and Reverse Engineering.

Components obtained by this kind of methods have loosely coupled semantic structure, which ensures low reuse cost, however, in these methods, component reusability and the ability to adapt to changes are unable to guarantee to keep in a high level, and, optimization on granularity basically does not involve, neither. Actually, the obtained components are all fixed in one or several granularity levels, and component designers even try to seek the balance between granularities of different components by intention [15], i.e., if a component has coarser granularity than others, it will be decomposed to decrease the granularity, which is widely divergent with current tendency, or conventions, of coarse-grained component reuse and runs in an opposite direction.

For the identification of coarse-grained entity components, we can compose multiple fine-grained entity components by inheritance and composition relationships between them to get coarse-grained ones [26], but for process components, there are no good methods at present yet.

2.3 Stability

Stability is originally a concept in Physics, and was defined as "the degree of the quality or attribute of being firm, steadfast and free from change or variation when outside conditions changes" [27]. Afterwards, it was introduced to the domain of System Theory, and was used to describe "the ability of a system, when kept under specified conditions, to maintain a stated property value within specified limits for a specified period of time"[27].

For an enterprise, its management patterns may be expressed by a set of business models. And along with the changes on internal requirements and external environments, business models may also change [28]. Different models have different stability. Some business elements seldom change, so they have higher stability, while other elements will frequent change, so they have lower stability. For example, the organization structures, position settings, execution rules in business processes, are all easily changed [29]. Stability could be measured as a volume of changes in the model needed to reflect changes in the reality. In the methodology of Six Sigma, the stability of a business process model is defined as the ability of the process to perform in a predictable manner over time, and is evaluated by the tool Run Chart [30].

In the field of software engineering, stability is used to describe the ability of a software artifact to keep unchanged along with the time, or the ability to adapt to changes by its flexible configuration mechanism [31][32]. In [33] a stability metrics for software architecture and components based on Case-based Reasoning (CBR) is presented, in which, the stability is measured by the data produced in the process of software evolution. Researchers have already obtained the conclusion that those components that are more frequently reused are more stable than the ones that are seldom reused [34].

So far, a majority of domain analysis methods do not involve the factor of "stability" in component design. This is because that there exists a basic hypothesis that all the commonalities in domain business models are all stable along with time [35]. But actually, this hypothesis is not true because any software artifacts will change more or less with time. Hamza and Fayad did some deep research on the stability of software systems, and they have proposed a Stability-Oriented Domain Analysis (SODA) method [32][35], in which the commonalities are classified into enduring ones and instable ones, accordingly, the software system is separated into three layers: core and essential *Enduring Business Themes* (EBTs)、seldom changed *Business Objects* (BOs) and frequently changed *Industrial Objects* (IOs). In this way, the final components have clear stability difference. But SODA does not suggest constructive strategies on component granularity optimization, neither.

3 Feature-oriented component model and granularity metrics

Feature modeling is the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a feature model. Feature-oriented method has been widely applied in the field of software reuse long time before, e.g., FODA[18], FORM [19], etc. Feature, as a tool, is used to depict the commonalities and variabilities among related business in a given business domain to form domain feature model for reuse. Reusable business models contain inherently more variability than concrete models and feature modeling is the key technique for identifying and capturing variability.

Because there exist an intrinsic consistency between domain features and the services provided by components, feature modeling method is also used to describe component models [12], and makes component models and domain models in a uniform semantics space, thus creates a direct mapping between them two. In addition, some of characteristics of feature, such as hierarchy, expansibility and multi-dimension, make it more suitable to express the variations in domain models and component models than other technologies.

In this section, a feature-oriented component model is presented. It is a semi-semantic component specification model, with emphasis on the representation of the commonalities and variabilities of business semantics, and dependencies between them. We try to use feature space as a tool to create a direct mapping between component and business models, i.e., business models and component models are uniformly expressed as feature space, and a component is regarded as a sub-space of business model's feature space.

First some taxonomy of feature and feature dependency are introduced, and then the feature-oriented business model is presented. Finally some metrics for measure component granularity, and relationships between component granularity and component reuse performance are discussed briefly.

3.1 Feature and feature dependency

Here we briefly introduce the concepts of feature and feature space in traditional feature modeling methods.

Definition 3.1. (feature[12]). Feature is an ontology that is used to describe the knowledge of external world, and is represented as "Terms" or "Concepts" describing the services supplied by a specific business domain.

Definition 3.2. (feature space[12]). Feature space is a semantics space composed with a set of related features in a specific domain, and dependencies between these features, denoted as Ω=<*F*, *D*>, where *F* is set of features, and *D* is set of feature dependencies between features in *F*.

Ω is usually represented as the form of a feature tree, in which, there exists one and only one root node f_{root} as the root feature of Ω, and the conjoint two features in this tree are respectively called parent feature and child feature. Each child feature specifies a portion of its parent feature's semantics. The relationship between father feature and child feature is a kind of aggregation that is used to describe the whole-part association (WPA) between features, and makes the feature space form a hierarchical structure. Denote child(*f*), parent(*f*), ancestor(*f*), descendant(*f*) and sibling(*f*) as *f*'s child feature set, parent feature set, ancestor feature set, descendant feature set and sibling feature set, respectively, and we have

$$\text{ancestor}(f)=\begin{cases}\bigcup\limits_{g\in\text{parent}(f)}\text{ancestor}(g), & f\neq f_{root}\\ \Phi, & f=f_{root}\end{cases}$$

$$\text{descendant}(f)=\begin{cases}\bigcup\limits_{g\in\text{child}(f)}\text{descendant}(g), & \text{child}(f)\neq\Phi\\ \Phi, & \text{child}(f)=\Phi\end{cases}$$

If a feature *f* has multiple parent features, then *f* is a common feature. We can construct a copy of f and all its decedent features for *f*'s every parent feature so as to ensure the feature tree satisfy the characteristics of a common tree.

Definition 3.3. (feature item). A feature Item is an instance of a specific feature, describes the feature's one possible value under a given business environment, and reflects the variability of the feature. Let *dom*(*f*) denote the set of all feature items of feature *f*, and is called the "*domain*" of *f*. For $\forall\tau\in dom(f)$, τ is called a value of *f*. A feature is an abstraction of all its feature items, and there exists a generalization-instantiation association (GIA) between a feature and its items.

Definition 3.4. (feature instantiation). Feature *f*'s instantiation is the process of selecting one feature item from its domain *dom*(*f*) to satisfy a requirement context *R*, denoted as $\tau_R(f)$. If the meaning of R is clear, $\tau(f)$ will be used instead of $\tau_R(f)$.

An instance of feature *f* can be denoted by the instances of its child features. For $\forall\tau\in dom(f)$, there must $\exists Y\subseteq child(f), Y=\{f_1, f_2,\ldots, f_n\}$, and for $\forall f_i\in Y$, there must at least exist one feature item $\tau_i\in dom(f_i)$ that makes τ is determined by $\tau_1,\tau_2,\ldots,\tau_n$ uniquely. Y is called the essential sub-feature set of τ, denoted as es_set(τ).

Now we extend the traditional feature theory by proposing the definition of feature dependency. A feature dependency (FD) describes the relationships between two related features in a feature space. According to the structural and semantic relationships between two features, FD can be classified into four types:

- Feature Integrity Dependency (FID)
- Feature Value Dependency (FVD)
- Feature Multi-Value Dependency (FMVD)
- Feature Semantics Dependency (FSD)

Definition 3.5. (FID). There exists FID between feature *f* and feature set *Y*, if and only if $Y\subseteq child(f)$, and for every feature item τ of *f*, τ uniquely determines a set $Y'\subseteq Y$ that satisfies $Y'\subseteq$es_set(τ), denoted as $f|\to Y$.

$f|\to Y$ can be refined into four types, denoted as $f|^M\to g$, $f|^O\to g$, $f|^S\to Y$, and $f|^T\to Y$, respectively.

- Mandatory FID: $f|^M\to g\Leftrightarrow\forall\tau\in dom(f)$, there must $g\in$es_set(τ);
- Optional FID: $f|^O\to g\Leftrightarrow\exists P,Q\subseteq dom(f), P\cup Q=dom(f), P\cap Q=\varnothing$, for $\forall\tau\in P$, there exists $g\in$es_set(τ), and for $\forall\tau\in Q$, there exists $g\notin$es_set(τ);
- Single selection FID: $f|^S\to Y\Leftrightarrow|Y|\leq|dom(f)|$; There exists an clustering $\{P_1, P_2,\ldots, P_n\}$ of *dom*(*f*), $n=|Y|$, $P_i\cap P_j=\varnothing$, $\cup_{i=1\ldots n}P_i=dom(f)$, and for $\forall P_i$, there exists one and only one $f_i\in Y$, that makes $\forall\tau\in P_i, f_i\in$es_set($\tau$), and for $\forall f_i'\in Y\backslash\{f_i\}$, $f_i'\notin$es_set(τ);
- Multiple selection FID: $f|^T\to Y\Leftrightarrow$There exists an clustering $\{P_1,P_2,\ldots,P_n\}$ of *dom*(*f*), which makes $P_i\cap P_j=\varnothing,\cup_{i=1\ldots n}P_i=dom(f)$, and for $\forall P_i,\exists Y'\subseteq Y$, for $\forall\tau\in P_i,\forall f_j\in Y'$, there exists $f_j\in$es_set(τ), and for $\forall f_j'\in Y\backslash Y'$, $f_j'\notin$es_set(τ).

FID can be regarded as the dependencies between the "Values" of parent feature and the "Type" of its child features, and is called "Value-Type" dependency, i.e., one feature item of parent feature determines which of sub-features are the essential parts of the parent feature. FID depicts the structural integrity relationships between parent and child features by the choice of whether a child feature should be contained in the parent feature under different circumstances. It is also the only feature dependency presented in traditional feature modeling methods. FID is kind of explicit dependency, and can be expressed by the structure of feature space.

Definition 3.6. (FVD) Suppose the feature set *X* and *Y* are two subsets of F in feature space Ω, and for arbitrary two instances t1, t2 in Ω's instance set $T(\Omega)$, if $t_1[X]=t_2[X]$ comes true, then $t_1[Y]=t_2[Y]$ is also true, then *Y* is feature value dependent on *X*, or *X* feature determines *Y*, denoted as $X\to Y$, i.e., one instance of X uniquely determines one instance of Y.

Definition 3.7. (FMVD). Suppose the feature set *X*, *Y* and *Z* are three subsets of *F* in feature space Ω, and $Z=F-X-Y$. The FMVD $X\to\to Y$ comes true if and only if for every identical instance of (*X*, *Z*) determines a set of *Y*'s instance, and these instances only lie on the instances of *X*, and does not relate to *Z*.

FVD and FMVD depict the restrictions that must be satisfied when different features are instantiated. The essence of them are same, they both express the dependencies between feature items of two set of features, and can be called "Value-Value" dependency, i.e., the instances of one feature set uniquely of multiply determine the instances of another feature set. They generally appear between sibling features.

Definition 3.8. (FSD). FSD refers to the semantics relationships between features. According to the types of features, FSD possibly have multiple types, e.g., the temporal constraints between business operation features, the association, inheritance and composition constraints between business object features, the Event-Condition-Action (ECA) constraints between business activity features. Generally, we use a predication P(X) to denote that features in X should satisfy P. The concrete expressions of P are determined considering the concrete constraint types.

FSD has no relationships with feature instantiation, and it depicts the semantics constraints between features. By the way, FVD, FMVD and FSD are implicit dependencies.

3.2 Feature-oriented component model

A business component defines a sub space of a specific business domain. The feature-oriented component model can be denoted as $C<cid, f_{root}, F, D, PS, RS>$, in which,

- cid is a unique identification of C;
- f_{root} is the root feature, F is the feature set, $\forall f \in F$, $dom(f) \geq 1$, and D is the set of feature dependencies between features in F;
- PS is the set of features that C provides, RS is the set of features that C requires, and PS, $RS \subseteq F$. Features in PS constitute the PROVIDE interface of C, and features in RS constitute the REQUIRE interface of C.

For a normalized component, it is often true that $PS=\{f_{root}\}$, i.e., C only provides its root feature, which has the coarsest granularity, to other components, and if C need to provide other features in F, these features had better be separated out and form new independent components.

Component reusability can be represented through the following two aspects according to four types of feature dependencies mentioned in section 3.1:

(1) Feature's variability: Only in some certain situations, one feature is an essential constituent of component feature space, and in other situations it is not. According to FID, features can be divided into four types, i.e., mandatory features, optional features, single-selection features and multiple-selection features;

(2) Feature item's variability: One feature can be instantiated as an arbitrary feature item contained in the feature's domain according to FVD/FMVD. The existence of FVD/FMVD, i.e., $X \rightarrow Y$ or $X \rightarrow\rightarrow Y$, makes the instantiation of X and Y cannot keep independent on each other, and it is necessary to instantiate features in Y according to the results of X's instantiation to ensure the validity of $X \rightarrow Y$ ($X \rightarrow\rightarrow Y$).

In fact, feature's variability can be transformed to feature value's variability in a simple way. For $\forall f$ which is a non-mandatory feature, by adding one null feature item $I_\varnothing$ into $dom(f)$, f can now be regarded as a mandatory feature approximately. If f should not be chosen in some circumstances, f can be considered to be instantiated as $I_\varnothing$.

Under the premise of all the feature dependencies in a component C, by choosing one specific feature item for every feature in C, C is instantiated and a set of "component instances" are obtained. Denote instance(C) as the set of all the component instances, and for $\forall f \in F$, $\forall t \in$ instance(C), denote $\rho(f, t)$ as the corresponding feature items of f in t. Component instantiation is carried out when the component is practically reused.

In **Fig.1** we present an example component that contains two fixed features f_1, f_2, three variable features f_3, f_4, f_5, and a root feature f_{root}. There exist feature dependencies $\{f_3\} \rightarrow \{f_4\}, \{f_3\} \rightarrow \{f_5\}$ between f_3, f_4, f_5, and $dom(f_1)=\{\tau_{31}, \tau_{32}, I_\varnothing\}$, $dom(f_2)=\{\tau_{41}, \tau_{41}\}$, $dom(f_3)=\{\tau_{51}, I_\varnothing\}$. Every feature item of f_3, f_4, f_5 has a specific implementation or a null implementation $\varnothing$.

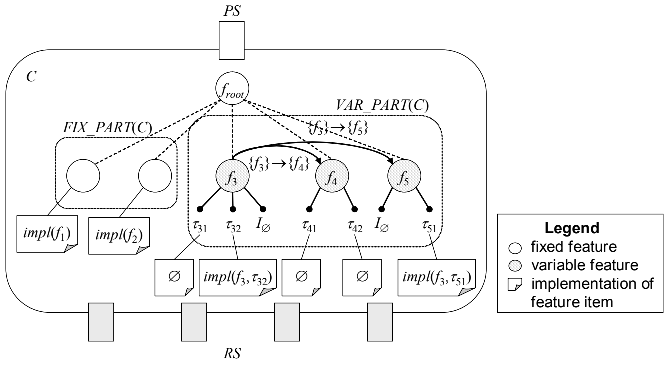

Fig.1 Feature-based component model and its variation mechanism

3.3 Component granularity

According to the general component model in last section, here we propose some definitions of component granularity and its metrics, in which feature's granularity is used to describe component granularity.

Definition 3.9. (feature's granularity level [12]). In the feature space of a specific domain, we use the difference between the layer of feature f and the *root* feature as f's granularity level, denoted as $GL(f)=layer(f)-layer(root)$. We have layer($root$)=0, $GL(root)$=1, and $layer(f)=layer(parent(f))+1$. Features in the same granularity level usually have the same semantics type, e.g., business activities, business operations, etc.

Definition 3.10. (feature's granularity) A feature f's granularity $G(f)$ is defined as the sum of all its child features' granularities, i.e.,

$$\mathrm{G}(f)=\begin{cases} \sum\limits_{f_i \in \mathrm{child}(f)} \mathrm{G}(f_i), & \mathrm{child}(f) \neq \Phi \\ 1, & \mathrm{child}(f) = \Phi \end{cases}$$

Because a component c can be regarded as a sub-space of domain feature space, c's granularity can be defined by the granularities of features contained in it.

In section 1 we have mentioned that the granularities of components in the same granularity level can be measured from the following two views: the number of functions that a component can provide outside by its interfaces, the number of sub entities contained in the component. They are called interface granularity and implementation granularity, respectively. From another view, component granularity can also be classified into absolute granularity and relative granularity. So we have four types of granularity, the definitions of which are as follows:

- *Interface Relative Granularity*: the number of features contained in PS(c), i.e.,
$$\mathrm{IRG}(c)=|\mathrm{PS}(c)|$$
- *Interface Absolute Granularity*: the mean value of granularity of all features in PS(c), i.e.,
$$\mathrm{IAG}(c)=\frac{1}{|\mathrm{PS}(c)|}\sum_{f \in \mathrm{PS}(c)} \mathrm{G}(f)$$

- *Implementation Relative Granularity*: the number of leaf features in components, i.e.,
$$\mathrm{NRG}(c)=\left|\{f \mid f\in \mathrm{F}(c), \mathrm{child}(f)=\Phi\}\right|$$
- *Implementation Absolute Granularity*: the granularity of f_{root}, i.e.,
$$\mathrm{NAG}(c)=\mathrm{G}(f_{root})$$

Relative granularity depicts the number of features a component provides or contains, but it doesn't depict the difference between these features' granularity. Absolute granularity considers this difference. For a generic component, interface granularity and implementation granularity is normally uniform, i.e., a component usually provides the feature with the coarsest granularity (root feature). If extra remarks are not added, the taxonomy "component granularity" referred in this paper means implementation absolute granularity NAG(*c*).

3.4 Relationships between granularity and component reusability

Granularity has great influence to component reusability, which is reflected in three aspects, i.e., reuse efficiency, composition cost and change cost. Firstly the definitions of the three properties are put forward.

Definition 3.11. (Reuse efficiency). Component *c*'s reuse efficiency REF(*c*) is defined as *c*'s contribution to the assembly of a domain feature space cooperating with other components, and can measured by the proportion of *c*'s feature space's size in the total domain feature space, denoted as

$$\mathrm{REF}(c)={|\mathrm{F}(c)|}\big/{|\mathrm{F}(\Omega)|}$$

It is easy to know that, the coarser *c*'s granularity is, the more contribution *c* can make to the whole domain feature space, and therefore the higher reuse efficiency *c* has.

Definition 3.12. (Composition cost) Component *c*'s composition cost CPC(*c*) is defined as the cost during the process that *c* is composed with other components to form the whole domain models. As can be foreseen, the coarser *c*'s granularity is, the few the times of composition with other components is, and the lower CPC(*c*) is.

Definition 3.13. (Change cost) Component *c*'s change cost CGC(*c*) is defined as the cost that *c* adjusts its structure or behaviors to satisfy different business requirements when *c* is reused in multiple business environments. In feature space, the change cost can be regarded as the cost to instantiate every feature contained in *c*, or, the cost to choose a right feature item for every feature.

Easily understood, the coarser *c*'s granularity is, and more easily the features contained in *c* changes, the higher CGC(*c*) is. Change cost is closely related to the stability property (will be defined in section 5.1) of features in *c*. In order to decrease change cost, we have to improve the stability of feature in *c*, or decrease *c*'s granularity. And in order to adequately utilize the advantages of coarse-grained components on reuse efficiency and composition cost, we may only adopt the approach of improving stability of features in components without losing the granularity.

One of the most important tasks during component identification is to seek the optimized granularity so as to decrease the change cost of coarse-grained components as much as possible under to guarantee that the reuse efficiency and composition cost both keep in an optimization level. In next section we will present the concept of dynamic granularity, and by acquiring components with dynamic granularities, we try to obtain the integrated optimization on component's reuse performance.

4 Mapping between business model and component model

As discussed above, the process of component reuse is the process to construct domain feature space using components. While a domain feature space is obtained from a set of business models. Here we firstly give some basic descriptions about business models, then discuss the mappings between domain feature space and component space.

4.1 Business model and its feature space

In the domain of enterprise information system, business models reflect the basic management patterns of the enterprise, and are the foundations for the design of enterprise information systems. Business models in the same domain may have some similarities between different industries and companies according to different manufacturing types, such as MTO, ETO, MTS, etc. These similar business models constitute a domain business model together.

There exists multiple views in business models, and every view contains several types of business elements. These elements can be classified into static elements and dynamic elements, the former of which includes domain, organization, position, business object, etc, and the latter of which includes domain flow, business process, business activities, etc, which depict the business flows between different departments or positions in enterprises. When we use these models to construct software components, the two types of business elements could be mapped to entity components and process components, respectively. Mechanisms for the two kinds of mappings are quite different. Mapping from static elements to entity components is relatively easier, and in literatures there exists many methods which have implemented this mapping, the main approach of which is by aggregation, i.e., grouping multiple closely related fine-grained elements into a coarse-grained entity component. Mapping from dynamic elements to process components seems a little more complex, the main idea of which is by decomposition, i.e., clustering a complex business process into some loosely-coupled sub-processes and map every one of them to a coarse-grained process component. In this paper we primarily discuss the identification and design of process components.

According to their semantics types, elements which constitute a business process model can be divided into several layers, i.e., process, sub-process, activity, operations, etc, and every layer is called a basic granularity layer. There exists composition relationships between elements in two neighboring layers, i.e., an element in an upper layer are composed of a set of elements in lower layer.

Therefore, a single business process forms a feature space, and can be regarded as a feature tree, in which the business process element is the root feature, and is decomposed gradually downwards into sub-process features, activity features and operation features. Business rules between activities or operations (such as ECA rules) can be expressed by a set of FSD. In the following, a business element e is accordingly denoted by a feature f in domain feature space.

Suppose a business process bp forms a feature space $\Omega(bp)$, there exists n layers $L_1, L_2, \ldots, L_n$, and denote F_i as the business elements (or features) in ist layer. For the element $f_{i+1,j}$ in layer F_{i+1}, there exists a set of elements $child(f_{i+1,j})=\{f_{i1}, f_{i2}, \ldots, f_{ik}\}$ in layer F_i that makes $f_{i+1,j}=\Re\{f_{i1}, f_{i2}, \ldots, f_{ik}\}$, where $\Re$ is a composition operator.

4.2 Basic design process of reusable components

A component provides specific services that support the implementation of one or several business models, therefore a component can be regarded as a part of business models. There exists a mapping relationship between business models and components, and can be expressed by the following formula:

$$C \equiv \text{Aggregate}(\text{Abstract}(\text{Decompose}(BM))) \qquad (1)$$

$$BM \equiv \text{Composite}(\text{Config/ Instantiate/ Adapt}(\text{Select}(C))) \qquad (2)$$

Expression (1) is the process of component design. By applying the operations of decomposition, abstraction and aggregation on a set of related business models BM, we get a set of components C. Three phases is included in this process, just as follows:

Decomposition. According to specific rules, decompose business models into a set of sub-models, and map every sub-model into a component. Some of decomposition rules include coupling-cohesion rule [6], data dependency rule and function dependency rule [2], etc.

Abstraction. Abstract those similar services with same business goals and different implementation mechanisms so that a component can be reused in multiple business situations to improve reusability. Practical abstraction techniques include dimensionality reduction, grouping, splitting, and intensionalization, etc [2].

An abstract component can implement multiple isomers of one specific business, i.e., different implementation styles of one business, which is called vertical abstraction. This kind of component usually provides different business logic or business rules to deal with the same business objects. For example, in a "Purchasing product arrival management" component, it can realize different business logic that is conduced by the different temporal order of the arrival of products and invoice. An abstract component can also implement the common functions used in multiple businesses, which is called horizontal abstract. This kind of components usually has a majority of commonalities and a minority of specific business logic/rules, and can deal with different business objects. For example, in a "Sales order management" component, it can deal with common orders, retail orders, long-term sales agreements, etc.

Aggregation. For those components which are often reused together, aggregate them to get a coarse-grained component so as to improve the reuse efficiency and reuse cost. Related aggregation techniques include Common Reuse Principle (CRP), inheritance/composition-based aggregation, etc [2][23].

In component design process, decomposition phase specifies the basic granularity of every component; abstraction phase has no effect on component granularity, and aggregation phase increases component granularity.

Expression (2) describes the process of component reuse, during which, appropriate components are firstly queried and distilled from library, and by configuration, instantiation and modification on these components, we can compose them into a system to satisfy the requirements described by business models. (1) and (2) are two reverse processes.

4.3 Mapping strategies between business models and component models

How to create the mapping between business models and component feature space, and decompose business models into a set of reusable components, is a key problem in component design process. This mapping should satisfy the properties of completeness and non-intersection. Completeness means that every business element in business models can be implemented by one or several components, denoted as

$$\begin{aligned} &\forall f \in \Omega(bp),\ \exists c_1, c_2, \ldots, c_n \in C,\ n \geq 1, \\ &\text{makes } f = \Re\left\{f_{11}, f_{12}, \ldots, f_{1k_1}, \ldots, f_{n1}, f_{n2}, \ldots, f_{nk_n}\right\}, \\ &\text{in which } f_{ij} \in \text{PS}(c_i),\ 1 \leq i \leq n,\ 1 \leq j \leq k_i \end{aligned} \qquad (3)$$

n=1 means that f is a feature that is provided by a single component; n>1 means that f must be implemented by the composition of features provided by multiple components $\{c_1, c_2, \ldots, c_n\}$.

Non-intersection means that for arbitrary two components c_i, c_j, their feature spaces should not intersect with each other, $\Omega(c_i)\cap\Omega(c_j)=\varnothing$, i.e., there do not exist any features that appear in two or more components at the same time.

Based on the conclusion above and comparisons between several existed component design methods in literatures, we classify the mapping into four types: SG-mapping, MG-mapping, IG-mapping and DG-mapping.

4.3.1 Single granularity layer mapping (SG-mapping)

In SG-Mapping, one granularity layer L_i is firstly specified, then every element f belonging to layer L_i in domain feature space (representing a specific domain business model) is directly mapped to a component, i.e., for $\forall f \in F(L_i)$, f is mapped to a component $c_{\{f\}}$, denoted as $f \rightarrow c_{\{f\}}$. In addition, f's all descendant elements descendant(f) are all contained in $c_{\{f\}}$, i.e., $F(c_{\{f\}})=\{f\}\cup$descendant(f); For all the elements whose layer are higher than f, they can be implemented by the composition of components obtained by SG-mapping. The mapping is shown in **Fig.2**.

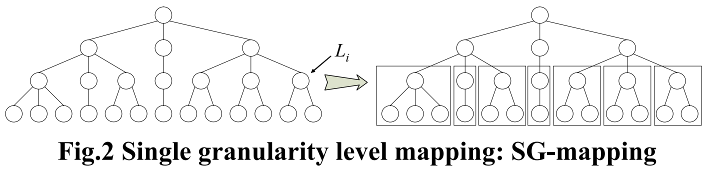

Fig.2 Single granularity level mapping: SG-mapping

SG-mapping is simple, but its shortcomings are also obvious, i.e., obtained components are all in the same granularity layer, which will lead to poor reusability. For example, if L_i is lower in domain feature space, these components are fine-grained with lower reuse efficiency and higher composition cost, but if L_i is higher, then those components are coarse-grained, but are possible to lead to poor change cost.

4.3.2 Multiple granularity layer mapping (MG-mapping)
MG-mapping is based on SG-mapping. It maps elements in multiple granularity layers synchronously and separately to realize multi-granularity component co-existences. For example, for two layers L_i, L_j (L_i<L_j), we apply SG-mapping separately on each element in L_i and L_j. For each element *g* in the layer below L_i, if *f* is a feature in layer L_i and $f \in$ancestor(*g*), then g is contained in the component $c_{\{f\}}$ obtained from *f* by SG-mapping. For each element between layer L_i and L_j, it can be implemented by composition of the components obtained from SG-mapping on L_i. For each element in layer higher than L_j, it can be implemented by the composition of components obtained from SG-mapping on L_j. MG-mapping is shown in **Fig.3**.

The primary benefit of MG-mapping is that granularities of components are more flexible, and components with different granularities can co-exist in the result component set. But this kind of flexibility is only limited into those basic granularity layers in business models.

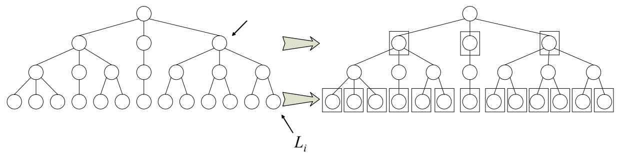

Fig.3 Multiple granularity level mapping: MG-mapping

4.3.3 Middle granularity layer mapping (IG-mapping)
A similar problem in SG-mapping and MG-mapping is that the final components' granularity is basically specified, that is to say, when a granularity layer is specified, components to be obtained are all in the same granularity layer of the corresponding business elements and cannot be changed.

The main idea of IG-mapping is, first specify one layer L_i, and according to the coupling degree between elements in this layer, cluster these elements to get multiple components. These components has granularity that is between L_i and L_j. For example, if we specify the business activity layer for IG-Mapping, every obtained component contains one or several business activities, as shown in **Fig.4**.

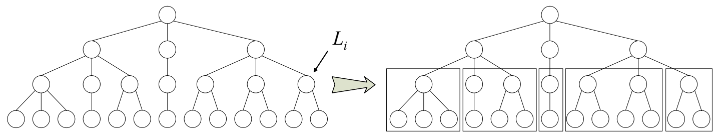

Fig.4 Middle granularity layer mapping: IG-mapping

For IG-mapping, it can also realize multiple middle granularity layer mapping (MIG-mapping) i.e., specify multiple layers, and cluster elements in every layers separately to form components.

IG-mapping realizes some flexibility on component granularity to some extent, but the flexibility is only limited between the specified granularity layer and its neighboring low layer.

4.3.4 Dynamic granularity layer mapping (DG-mapping)
SG-mapping, MG-mapping and IG-mapping have the same characteristics that the components' granularity is basically specified when one or several granularity layers are chosen before the mapping, and the flexibility on granularity is very small. In addition, all the three mappings do not consider the semantics properties of every business element, so the granularity of components indeed has no essential relations with the characteristics of these elements, which will lead to poor reusability.

It is obvious that the mapping should produce components with dynamic granularity, which is called DG-mapping. If we combine the above three mapping strategies together, for arbitrary feature *f* in arbitrary layer L_i, when *f* is mapped to component space, there are three possible strategies:

- Directly mapped to a component;
- Mapped as a part of a component
- Composed by a set of components

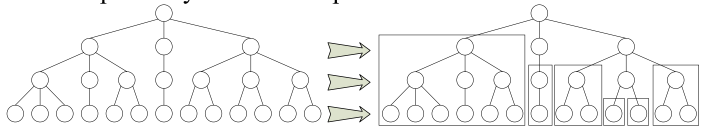

Fig.5 Dynamic granularity layer mapping

The key of DG-mapping is the mapping principle, i.e., according to which specified principles to map one element to component space, and how to ensure high reusability, as shown in Fig.5. We will present a component design method based on business model stability, in which business elements' stability is evaluated to determine which of the three mapping strategies is used in DG-mapping for each element.

5 Stability and stability dependency

The instability of business models is expressed in two aspects: instability on time axis, instability between different industries or enterprises. The former behaves that, a business is running with current pattern, but after a period of time, it changes to another patterns. The latter behaves that, different industries/enterprises have different patterns for a same business. The stability of a business model is defined as the degree to keep stable for the business model both in time and in space, and can be evaluated by the degree of frequently changing of this model.

In this section we firstly discuss the relationships between model stability and component granularity, and then we present a metrics for model stability from two aspects: stability of business elements, stability of relationships between elements.

5.1 Relationships between business model stability and component granularity

As discussed above, components are obtained from business models by a specific mapping strategy. An instable business model needs frequent changes, but the corresponding software system that supports the running of this model should not be changed so frequently. If changes happen after the software system has been normally running, modifications on it will have to lead to frequently interruptions and restarts, which will also cause huge cost.

Therefore, the components which are acquired from business models by mapping and implement software systems by composition should have to accommodate to these changes, i.e., components should be considered as an isolated area, or a cache, between instable business models and stable software systems in order to avoid the frequent changes on software systems. This requires that a component itself should have the ability of adapt to changes, i.e., it can satisfy different business requirements by configuration, and will not or rarely influence the corresponding software system. If changes on business models cannot be abstracted and encapsulated into component in a configurable form (i.e., business rules, parameters, etc), the granularity of components has to be decreased, and these changes will have to be realized in reuse process by developers for concrete business circumstances, otherwise, components have to face to the huge change cost of frequent changes when reused.

From the analysis above we can know, component granularity is closely related to the stability of business models. The more stable the business model is, the few changes it will have, then, the few changes the corresponding components need to adapt to, and the coarser granularity the component will have. So does it on the contrary, i.e., the less stable the business model is, the more changes it will have, then the more difficult the corresponding components adapt to these changes, so we have to decompose the business model into some finer-grained components.

Therefore, for stable business models, they can be directly mapped to coarse-grained components. For instable business models, they should be decomposed into a set of sub-models, map those stable sub-models into components, and sequentially decompose those instable sub-models, until every obtained sub-model are stable enough and all the business elements in this model have been mapped to components.

5.2 Stability and stability metrics

A business element's stability is closely related to the number of its isomers. Different isomers realize the same business goals, but with different implementation strategies. If we regard a business element as a feature in feature space, then the set of all its isomers is the instance set of the corresponding feature, and every isomer can be denoted as a feature instance. For their inner structures, different isomers include different child feature set, or same child feature set but instantiated to different feature items.

Definition 5.1 (Isomer) A business element f's isomers are defined as the instances of f. All of them have the same input/output, but with different implementation, denoted as $R(f)=\{\tau_1, \tau_2,.., \tau_n\}$.

For example, for a business activity element "Production planning", its input is sales order information, and its output is production plans, but according to different manufacturing types, such as ATO, ETO, MTO, etc, the inner planning algorithms are quite different, so it has a set of isomers. In **Fig.6** we present an example of isomers, in which, (a), (b) and (c) describe three isomers τ_1, τ_2, τ_3 of f, the corresponding child feature set are $\{u, v, w\}$,$\{u, v\}$,$\{u, v\}$, and instantiated to $\{u_1, v_1, w_1\}$,$\{u_1, v_2\}$,$\{u_2, v_3\}$, respectively. (d) shows the whole feature tree of f.

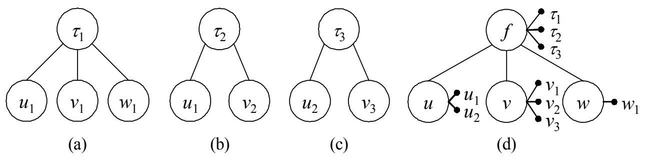

Fig.6 Examples of isomers

The stability S(f) of business element f, is related with the following three factors: number of isomers N(f), distribution on reuse frequency of isomers P(f), and similarity between isomers L(f).

5.2.1 Number of isomers

The larger the number of isomers N(f) is, the more frequently f will change, and the less stability it has. The range of N(f) is $[1, +\infty)$ and N(f)=| instance(f) |.

5.2.2 Distribution on reuse frequency of isomers: stability entropy

N(f) measures the stability of business elements from one aspect, but the relationship between N(f) and S(f) is not absolutely linear. S(f) is also related to the distribution on reuse frequency of isomers. We use "stability entropy" to measure it.

Definition 5.2 (reuse frequency) A business element f's one isomer τ's reuse frequency, denoted as $\rho(\tau_i)$, is defined as the proportion of τ's reuse times in f's total reuse times in a period of time, and satisfies $0\le\rho(\tau_i)\le1$.

$\rho(\tau_i)$ could be gained from the reuse statistical data in a period of time, which is usually 6 to 24 months, and could comparatively reflect the degree of frequent reuse for every isomers factually.

Definition 5.3 (stability entropy) Suppose an element f's isomer set is $R(f)=\{\tau_1, \tau_2, ..., \tau_n\}$, for $\forall\,\tau_i\in R(f)$, its reuse frequency is $\rho(\tau_i)$, then f's stability entropy is denoted as

$$\mathrm{P}(f)=-\sum_{i=1}^{n}\rho(\tau_i)\log\rho(\tau_i) \tag{4}$$

A larger P(f) indicates that the distribution on reuse frequency of different isomers is more balanced, and f is less stable because f has more chances to be frequently switched between its isomers. A smaller P(f) indicates that the distribution is more lopsided, and f is more stable. The range of P(f) is $[0, +\infty)$. If N(f)=1, then there must exist $P(f)$=0, which indicates that f has only one implementation and it is absolute stable.

5.2.3 Similarity between isomers

Even if N(f) and P(f) are both very large, f is still possible to be stable. This is because that stability is also related to the similarity between isomers.

Definition 5.4 (similarity between isomers) The similarity L(f) is defined as the degree of similarity between the inner implementation of all its isomers. The more similar between isomers, the less difference between them, and the more stability f has. The range of L(f) is [0,1].

The metrics to evaluate L(f) can be separately discussed according to f's type: atomic element and complex element. Atomic elements refer to those elements that are in the bottom of domain feature space and cannot be decomposed any more (leaf features), e.g., business operations. Complex elements refer to those elements that can be decomposed into a set of child elements.

For an atomic element, we can simply get the similarity of its isomers by estimation. For a complex element, its similarity between isomers can be measured by the synthesis of similarities between every child elements and relationships between these child elements. For the latter, because relationships can be described by business rules to realize configurability, when evaluating similarity of complex elements, we ignore it and only consider the similarities between child elements. We use algorithm 1 to calculate the similarity between isomers of complex elements.

In addition, although some elements are not atomic, if it is unnecessary to be decomposed or is difficult to be decomposed, they can also be considered as atomic elements e.g., those business activities that is constituted of complex algorithms. If an atomic element has N(*f*)=1, then its similarity L(*f*)=1.

Algorithm 1

Input: a complex element *f* and its isomer set $R(f)=\{\tau_1, \tau_2, ..., \tau_n\}$, along with every isomer's child element set child(τ_i) and reuse frequency $\rho(\tau_i)$.

Output: the similarity between isomers of *f*.

```
1) Construct $f$'s feature tree $T(f)$ with the root node as $f$, and using the set $\bigcup_{i=1}^{n}\mathrm{child}(\tau_i)$, with all the same appearances of each
   child element merged, as the leaf node set child($f$);
2) For $\forall f_i \in \mathrm{child}(f)$, if $f_i$ is an atomic element, estimate its similarity $L(f_i)$; if it is a complex element, recursively use Algorithm 1 to calculate $L(f_i)$;
3) For $\forall f_i \in \mathrm{child}(f)$, calculate its reuse frequency $\rho(f_i)$. Suppose $f_i$ appear in $f$'s several isomers $\{\tau_{i1}, \tau_{i2}, ..., \tau_{ik}\}$, then
   $\rho(f_i)=\sum_{i=1}^{k}\rho(\tau_{ij})$;
4) Calculate $f$'s similarity by $\mathrm{L}(f)=\min_{f_i\in\mathrm{child}(f)}(\mathrm{L}(f_i)\times\rho(f_i))$.
```

Using N(*f*), P(*f*) and L(*f*), we form the stability metrics S(*f*) as

$$\mathrm{S}(f)=1-\exp\left(-\frac{\mathrm{L}(f)}{\mathrm{N}(f)\times\mathrm{P}(f)}\right) \tag{5}$$

where $S(f)\in[0, 1]$. When *f* is in the most stable state, i.e., *f* has only one isomer, N(*f*)=1, P(*f*)=0, L(*f*)=1, S(*f*)=1. If *f* is in the most instable state, $N(f)\to+\infty$, $P(f)\to+\infty$, $L(f)\to 0$, therefore $S(f)\to 0$.

Theorem 1: In a business model, the stability of parent element is not larger than the stability of any of its child elements.

Proof: Suppose *f*'s isomer set is $\{\tau_1, \tau_2, ..., \tau_n\}$, N(*f*)=*n*, and for every τ_i, its reuse frequency is $\rho(\tau_i)$, *f*'s child element set is child(*f*)= $\{f_1, f_2, ..., f_k\}$, $\forall f_i\in$child(*f*), f_i's isomer set is $\{\tau_{i1}, \tau_{i2}, ..., \tau_{im_i}\}$, with the reuse frequency $\rho(\tau_{ij})$ respectively.

For *f*, *f*'s every child element f_i's every isomer τ_{ij} should have to appear in at least one isomer of *f*, and at the same time, *f*'s one isomer can only contain one isomer of f_i. So according to pigeonhole principle, if $N(f)<N(f_i)$, there exist at least some isomers of f_i with the number $N(f_i)-N(f)$ that cannot appear in *f*'s all isomers. This is impossible. So we have $N(f)\geq N(f_i)$.

According to $\rho(f_i)=\sum_{i=1}^{k}\rho(\tau_{ij})$ in step (3) in Algorithm 1, we know that compared with *f*, the distribution on reuse frequency of e_i's isomers is more centralized. According to Formula (3), we know $P(f)\geq P(f_i)$;

Again, according to $\mathrm{L}(f)=\min_{f_i\in\mathrm{child}(f)}(\mathrm{L}(f_i)\times\rho(f_i))$ in Step 4 in Algorithm 1, we know $L(f)\leq L(f_i)$;

According to Formula (5) and above conclusions, we can get

$$\mathrm{S}(f)=1-\exp\left(-\frac{\mathrm{L}(f)}{\mathrm{N}(f)\times\mathrm{P}(f)}\right)\leq 1-\exp\left(-\frac{\mathrm{L}(f_i)}{\mathrm{N}(f_i)\times\mathrm{P}(f_i)}\right)=\mathrm{S}(f_i),$$

namely, $S(f)\leq S(f_i)$.

5.3 Stability Dependency

In section 5.2 we have discussed the stability of business elements; here we emphatically discuss the stability of relationships between business elements. Because this kind of relationships exists between two elements, the stability of relationships actually expresses the degree of coalescent of the two elements, or is called "degree of stability dependency". The more stability a relationship between two elements has, the less easily it changes; therefore, the coalescent of the two elements is more stable and less easy to change.

Definition 5.5 (stability dependency, SD). For arbitrary two elements f_i, f_j, if the change on f_i will lead to the change on f_j, then we call f_j is stability dependent on f_i, or f_i stability determines f_j, denoted as $f_i \Rightarrow f_j$. The probability that f_j changes when f_i changes, is called the degree of stability dependency, denoted as SD(f_i, f_j). SD(f_i, f_j) is a condition probability , SD(f_i, f_j)=1 denotes that the changes on f_i is sure to trigger the changes on f_j, and SD(f_i, f_j)=0 denotes that the changes on f_i takes no effect on f_j and there exists no stability dependency between them.

Stability dependency depicts the dependency relationships between two business elements' stability, or in reverse, the change probability. The larger SD(f_i, f_j) is, the larger probability the two elements changes together, and therefore, they have higher semantics coupling.

As discussed previously, in a feature space there exist four types of feature dependencies (FD). Except FID which does not relate to stability dependency, the other three types of FD are the sources that will lead to stability dependency, so the degree of SD can also be evaluated by them both:

- If the type of FD between two features f_i, f_j is FVD or FMVD, then the degree of SD produced by FVD/FMVD between them is 1;
- If the type of FD between f_i, f_j is FSD, then the degree of SD between them is closely related to the complexity of data structure or function structure containing in this FSD, CPX(FSD(f_i, f_j)).

We can use the following formula to measure SD(f_i, f_j):

$$\mathrm{SD}(f_i, f_j)=\begin{cases}1, & \text{there exist FVD/FMVD between } f_i \text{ and } f_j\\ \mathrm{CPX}(\mathrm{FSD}(f_i, f_j)), & \text{there exist FSD between } f_i \text{ and } f_j\end{cases} \tag{6}$$

Many methods exist in literatures to measure the complexity of data structure or function structure, and we can directly use them to measure CPX(FSD(f_i, f_j)). We will not discuss it here for the limited space.

6 Component Design Method Based on Model Stability

In this section we study a component design process based on business model stability, in which, according to the stability of every business element, we use different mapping strategies to map it into component space and finally get a set of component with dynamic granularities. In the mapping process, we introduce the concept "Most Stable Set", and after finding every MS-set in current business model, we map all the elements in MS-set into one component and delete these elements from the business models. Repeat the process until all the elements have been mapped to components.

6.1 Stability set

Definition 5.6 (stable set in the same granularity layer, GLST) One business model *bm*'s one GLST *glst* is constituted by a set of business elements in *bm*, and they satisfy the following constraints:

1) All the elements in *glst* are in the same granularity layer in the feature space of *bm*;
2) For $\forall f \in glst$, $S(f) \geq \psi$ (ψ is a gate value);
3) For $\forall f \in glst$, there must at least $\exists g \in glst$ that makes $SD(f, g) \geq \xi$ (ξ is a gate value).

Definition 5.7 (stable set) One stable set *st* of a business model *bm* is constituted of a set of business elements in *bm*, and they satisfy the following constraints:

1) $\forall f_i \in st$, $S(f_i) \geq \psi$;
2) $\forall f_i \in st$, $descendant(f_i) \subseteq st$;
3) Suppose that F_r is the set of elements that for $\forall f_i \in F_r$, f_i satisfies $parent(f_i) \notin F_r$, then for $\forall f_i, f_j \in F_r$, $parent(f_i) = parent(f_j)$. F_r is called the "root element set" of *st*;
4) F_r is a GLST of *bm*.

From theorem 1 we can know that, if $S(f_i) \geq \psi$, then for $\forall f_j \in descendant(f_i)$, there must exist $S(f_j) \geq \psi$, so condition (2) is consistent with condition (1). *st* is also called "ψ-stable set", and when ψ=1, *st* is a "total stable set".

Definition 5.8 (Most stable set, MSS) One ψ-stable set *st* of business model *bm* is called a ψ-most stable set, if and only if there doesn't exist an element $f_i \in \Omega(bm)$, $f_i \notin st$, and after adding f_i and *decendent*(f_i) into *st*, *st* is still a MSS of *bm*.

6.3 The Algorithm

Algorithm 2

Input: domain business model *bm*, stability *S*(*f*) of each business element *f* in *bm*, and gate value ψ and ξ

Output: Component set *C*

```
1) Let $st=\emptyset, C=\emptyset$;
2) For every $\forall f \in \Omega(bm)$, set flag($f$)=0;
3) Choose an arbitrary element $f$ that satisfies flag($f$)=NO_DEALT and
   parent($f$)$\neq\emptyset$ and flag(parent($f$))= INSTABLE;
   If $S(f) \in [\psi,1]$ {
       Add $f$ and all its descendant elements descendant($f$) into $st$;
       For $\forall f' \in \{f\} \cup$descendant($f$) {
           Set flag($f'$)=DONE
       }
       Go to step (4)
   }
   If $S(f) \in [0,\psi]$ {
       Let $flag(f')$= INSTABLE
       Continue to execute step (3)
       If there does not exist such $f$ any more {
           Go to step (6);
       }
   }
4) If sibling($f$)$\neq\emptyset$ {
       Choose $\forall f' \in$sibling($f$) that satisfies flag($f'$)=NO_DEALT
           If $S(f') \in [\psi,1]$ and $\exists f'' \in st$ that satisfies $SD(f', f'') \geq \xi$ {
               Add $f'$ and descendant($f'$) into $st$
               Set flag DONE for these elements
           }
           If $S(f') \in [0,\psi]$ {
               Let flag($f'$)=INSTABLE
           }
   }
   Repeat step (4), until there doesn't exist any $f'$ in sibling($f$) that has
   flag NO_DEALT;
5) If $st\neq\emptyset$ {
           Map all the elements in $st$ into one component $c$
           Add $c$ to $C$
           Set $st=\emptyset$, return to step (3)
   }
   If $st=\emptyset$, directly go to step (3);
6) Choose $\forall f \in \Omega(bm)$ that satisfies flag($f$)=INSTABLE and descen-
   dant($f$)=$\emptyset$ {
           Directly map $f$ into a component $c$
           Add $c$ to $C$
           Let flag($f$)=DONE
   }
   Repeat step (6) until there does not exist any $e$ that satisfies such con-
   ditions;
7) Now all the elements in $\Omega(bm)$ has flag DONE, the algorithm ends,
   and $C$ is the final component set.
```

Here are some explanations about Algorithm 2:

- The set *st* that is obtained in every loop from step (3) to step (5) is a most stable set of *bm* (will be proved in Theorem 2), and is mapped to a component. In step 6, all the atomic elements which cannot be further decomposed and whose stability is lower than ψ is also mapped to a component.
- During the process, flag(*f*)=NO_DEALT means that *f* has not been dealt with yet, flag(*f*)=INSTABLE means that *f*'s stability is lower than ψ and cannot be directly mapped to component, and flag(*f*)=DONE means that *f* has been mapped to a component or a part of a component.
- ψ and ξ are two gate values that have great influence to the granularity of final components: ψ=1 means that only those totally stable elements can be directly mapped to component. The smaller ψ is, the coarser granularity the components will have, and the less stable these components are. Similar, the larger ξ is, the closer stability dependency between business elements in one component.

Theorem 2: In Algorithm 2, the set *st* that is obtained in every loop from step (3) to step (5) is a most stable set of *bm*.

Proof: From Algorithm 2 it is easy to validate that all the elements in *st* satisfy the constraints in Definition 5.6, 5.7 and 5.8, and we will not present the prove in details.

Theorem 3: the component set obtained by Algorithm 2 satisfies non-intersection and completeness.

Proof: During the execution process of Algorithm2, $\forall f \in \Omega(bm)$, only when *f* is tagged as DONE, is it indicated that *f* has been mapped as a component or part of a component. According to the conditions for feature selection in step (3) and (6), if *f* is tagged as DONE, then *f* will not be selected for further mapping. Therefore *f* will not appear synchronously in the feature space of two different components. So the characteristics of non-interaction come true.

Now we prove the completeness. In algorithm 2, features in Ω(*bm*) are dealt top-down according to their levels in Ω(*bm*), and when dealing with a feature *f*, we first calculate its stability: if it is stable, then *f* and *f*'s all descendants will be encapsulated into a component; if f is instable, then it is tagged as INSTABLE. Therefore, for tags in features in Ω(*bm*), there are only two possibilities: INSTBLE and DONE, and from step (6) we know that leaf feature's tag may only be DONE.

According to step (5) and (6), features with tag DONE have already been mapped into a component. For a feature *f* with tag INSTABLE, child(*f*) may be divided into to two sub-sets, namely, DONE_SET(*f*) and INSTABLE_SET(*f*), the former of which contains all features that are tagged as DONE, and the latter INSTABLE. If we use the same style to divide child set of each feature in INSTABLE_SET(*f*) recursively, until each feature with tag INSTABLE does not further contain INSTBLE_SET in its child feature set(this is possible, because all the leaf features have been tagged as DONE, and the final child only contains such leaf features). According to this process, *f* may finally be represented as a composition of features with tag DONE, and each of these features is sure to be the *root feature* of a specific component of *C*, therefore, *f* may be implemented by these components; accordingly, the completeness comes true.

6.4 A Case

Different component identification methods have different application scenarios, identification goals and emphasis, therefore the reuse performance of final obtained components also have comparatively diversity. In Table 1 we give some qualitative comparisons between our approach and traditional component identification methods in literatures.

Table 1 Comparisons between STCIM and various component identification methods in literatures

		Domain analysis based methods	Clustering based methods	CRUD-based methods	Other methods	STCIM
Input model forms		Domain models	UML Use case diagram, class diagram, activity diagram, etc	UML Use case diagram, Class diagram, Activity diagram, Event models, etc	Goal Decomposition Model; Business process models	Domain feature-based models
Applications domains		Various components	Entity components; Process components	Entity components; Process components	Mainly process components	Various components
Identification goals	Reusability	High	N/A	N/A	High	High
	Reuse cost	N/A	Low	Low	High	Low
	Reuse efficiency	N/A	N/A	N/A	Low	High
	Stability	N/A	N/A	N/A	N/A	Separation of Concerns
	Granularity	N/A	In pursuit of granularity balance; usually fine-grained components	N/A	Usually in pursuit of coarse-grained components	In pursuit of coarse-grained components as far as possible under the guarantee of high stability and cohesion/coupling
	Cohesion	N/A	High	High	N/A	High
	Coupling	N/A	Low	Low	N/A	Low
Mapping strategies		SG-Mapping MG-Mapping	IG-Mapping	IG-Mapping	IG-Mapping	DG-Mapping
Typical methods		FODA[12] JadeBird[20]	Cohesion-Coupling method [6] Clustering Point based method (CPM)[9]	COMO[14] O2BC[15]	Goal-driven method (GDM)[21]	STCIM

More, in practice, we have applied STCIM in the design and implementation of Enterprise Resource Planning (ERP) system and have got satisfactory results, which validate the effectiveness of STCIM. Here we show a simple example.

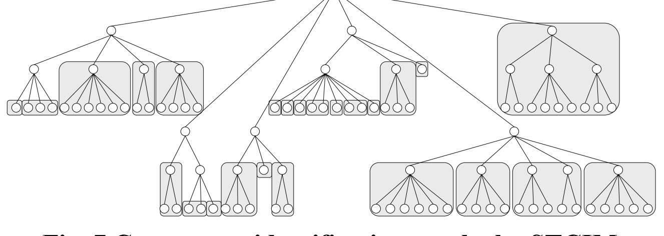
Fig. 7 Component identification results by STCIM

Fig. 7 (in which a circle represents a features and a rectangle represents a component, and due to limited space, we will not list the concrete meanings of the feature space) is the result of applying STIM on *PUOCUREMENT* domain in ERP. As a comparison, we select FODA[12], CPM[9], COMO[14] and GDM[21] and apply them on the same case. The final results of these methods are shown in Fig. 8, Fig. 9, Fig. 10 and Fig. 11.

After applying STCIM, we have evaluated the performance of the final component set by the five methods, as shown in Table 2, from which we may see that STIM adequately consider the optimization on stability and granularity under the guarantee that other reusability metrics do not deteriorate to much, therefore the synthetical performance of component set are better than other methods.

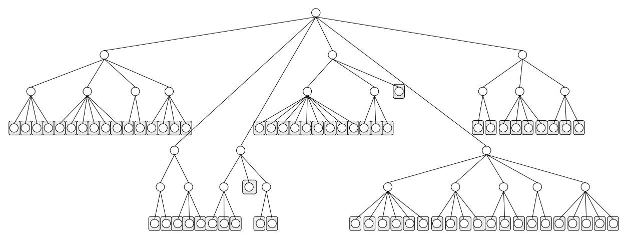

Fig. 8 Component identification results by FODA/FORM method

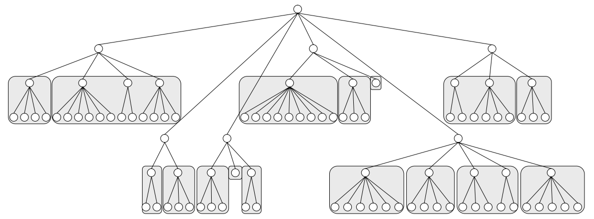

Fig. 9 Component identification results by COMO method

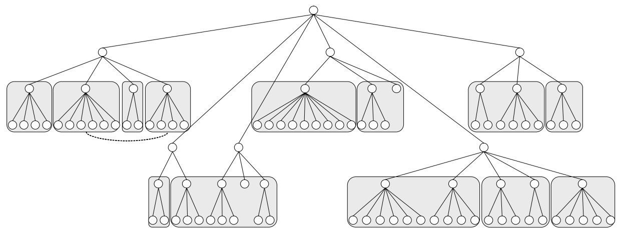

Fig. 10 Component identification results by core entity method

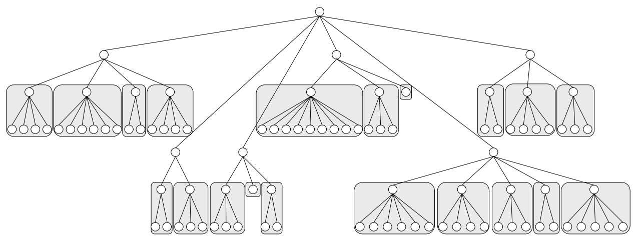

Fig. 11 Component identification results by goal-driven method

Table 2 Comparisons between component performance metrics of various algorithms

	FODA	CPM	COMO	GDM	STCIM
Component Number	69	12	16	20	25
Granularity	1	5.75	4.31	3.45	2.76
Reusability	1	0.37	0.22	0.50	0.76
Reuse Efficiency	0.01	0.08	0.06	0.05	0.04
Instantiation Cost	1.28	7.36	5.52	4.42	3.53
Composition Cost	8.45	4.36	3.20	5.01	6.92
Change Cost	6.84	20.04	25.21	12.26	10.32
Total Reuse Cost	16.57	31.76	33.93	21.69	20.77
Change Closeness	0.15	0.28	0.20	0.41	0.68
Stability	0.40	0.28	0.23	0.47	0.82
Cohesion	1.00	0.79	0.82	0.76	0.88
Coupling	0.58	0.24	0.21	0.44	0.27
Synthetical Performance	0.14	0.68	0.23	1.06	6.05

7 Conclusions

In this paper, aiming at the problem that current component identification methods lack of the optimization on component granularity, we present a new, dynamic granularity oriented and stability-based component identification method, STCIM. By calculating the stability of business models, we find a most stable set every time and map it to a component, accordingly realize the dynamic granularity and ensure the optimization on component reusability under the guarantee of coarser granularity as far as possible.

In conclusion, there are several innovations in the method presented in this paper, as follows:

- With the assistance of feature space as a tool, create the mapping between business model space and component model space, with four different mapping strategies presented correspondingly, to get components with dynamic granularity.
- Make clear the relationship between the stability of business models, component reusability and component granularity.
- Present a metrics for business model stability from three aspects: number of isomers of business elements, distribution on reuse frequency of isomers, and degree of similarity between isomers.
- Present a method for component identification based on business model stability, in which, find the most stable sets of business models and map them into components with dynamic granularities.

Future works includes: further research on the relationships between business model stability, component granularity and component reusability theoretically, design and implement a tool assisting components automatic identification from a set of similar business models, etc.

Acknowledgments

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

References

[1] Szyperski, C.(2002): *Component software: Beyond Object–Oriented Programming (2nd)*. Addison – Wesley, New York, NY.

[2] Mili, H., A.Mili, S.Yacoub, and E. Addy(2002): *Reuse–Based Software Engineering: Techniques, Organization, and Controls*. John Wiley and Sons Ltd., New York, NY.

[3] Papazoglou, M.P. and D. Georgakopoulos(2003): Service–Oriented Computing. Communications of the ACM. 46, 10, pp.25–28.

[4] Cioch, F.A., J.M. Brabbs, and L. Sieh(2000): The impact of software architecture reuse on development processes and standards. *The Journal of Systems and Software*. 50, 3, 221–236.

[5] HELTON, D(1998): The Impact of Large–Scale Component and Framework Application Development on Business. In *Proceedings of the 3rd International Workshop on Component–Oriented Programming*. Technical Report 17/99, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby. 163–164.

[6] Lee J.K., S.J. Jung, S.D. Kim, W. Hyun, and D.H. Han(2001): Component Identification Method with Coupling and Cohesion. In *Proceedings of 8th Asia－Pacific Software Engineering Conference*. IEEE Computer Society, 79–86.

[7] Stojanovic, Z., A.N.W. Dahanayake, and H.G. Sol(2004): Modeling and Design of Service-Oriented Architecture. In *Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics: Impacts of Emergence Cybernetics and Human-Machine Systems*. IEEE Computer Society, 4147–4151.

[8] Ali, A(2001): A Domain–Language Approach to Designing Dynamic Enterprise Component–Based Architectures to Support Business Services. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object–Oriented Languages and Systems (TOOLS39)*. IEEE Computer Society, 130–142.

[9] Roscaa, D. and C. Wild(2002): Towards a flexible deployment of business rules. *Expert Systems with Applications*. 23, 4, 385–394.

[10] Herzum, P. and O. Sims(1999): *Business Component Factory*. John Wiley&Sons, Inc. New York, NY.

[11] Somjit, A. and B. Dentcho(2003): Development of industrial information systems on the Web using business components. *Computer in Industry*. 50, 2, 231–250.

[12] Jia, Y(2002): *The evolutionary component–based software reuse approach*. PhD Dissertation, Graduation School of Chinese Academy of Sciences.

[13] Ai, P(2002): *Research on the Formal Method of Description for the Flexible Component Composition and its Application to the Water Resource Domain*. PhD Dissertation, Hohai University.

[14] Tang, C.J(2001): *A software synthesis methodology for developing component–based applications*. PhD Dissertation of Syracuse University.

[15] Xu, W., B.L. Yin, and Z.Y. Li(2003): Research on the business component design of enterprise information system. *Journal of Software*. 14, 7, 1213–1220.

[16] Harsu, M(2002): *A survey on domain engineering*. Report 31, Institute of Software Systems, Tampere University of Technology.

[17] SEI(2005): *Domain Engineering: A Model–Based Approach*. http://www.sei.cmu.edu/domain-engineering /domain_ engineering.html.

[18] Kang, K.C., S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson(1990): *Feature–oriented domain analysis(FODA) feasibility study*. Technical Report CMU/SEI–90–TR–021, Software Engineering Institute, Carnegie–Mellon University.

[19] Kang, K.C., S. Kim, J. Lee, and K. Lee(1999): Feature–oriented engineering of PBX software for adaptability and reuseability. *Software — Practice and Experience*. 29, 10, 875–896.

[20] Simos, M., D. Creps, C. Klingler, L. Levine, and D. Allemang(1996): *Organization domain modeling (ODM) guidebook, version 2.0*. Technical Report STARS–VC–A025/001/00, Lockheed Martin Tactical Defence Systems.

[21] Jacobson, I., M. Griss, and P. Jonsson(1997): S*oftware Reuse: Architecture,Process and Organization for Business Success*. Addison–Wesley, New York, NY.

[22] OMG(2003): *MDA Guide Version 1.0.1*, http://www.omg.org/cgi–bin/doc?omg/03–06–01.

[23] Martin, R.C(2002): *Agile software development: principles, patterns, and practices*. Prentice Hall, New York, NY.

[24] Lee, S.D. and Y.J. Yang(1998): COMO: A UML–based component development methodology. In P*roceedings of the 6th Asia Pacific Software Engineering Conference.* Takamatsu: IEEE Computer Society, 54–63

[25] Gansan, R. and S. Senguota(2001): O2BC: A technique for the design of component–based applications. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object–Oriented Language and Systems*. IEEE Computer Society, 46–55.

[26] Li, G. and M.Z. Jin(2000): A design method for reusable components. *Journal of Computer Research & Development*. 37, 5, 609–615.

[27] Online Dictionary(2005): http://www.seslisozluk.com/?word=stability.

[28] Alli, A., Z. Hussein, and A. James(2002):. Externalizing Component Manners to Achieve Greater Maintainability through a Highly Re–configurable Architectural Style. In *Proceedings of 2002 International Conference on Software Maintenance*. IEEE Computer Society, 628–637.

[29] Fayad, M.E(2002): Accomplishing Software Stability. *Communications of the ACM*. 45, 1, 111–115.

[30] ISIXSIGMA LLC(2005): http://www.isixsigma.com/dictionary/Process_Stability–453.htm.

[31] Bahsoon, R. and W. Emmrich(2004): Evaluating architectural stability with real options theory. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society, 11–14.

[32] Fayad, M.E(2002): How to Deal with Software Stability. *Communications of the ACM*. 45, 4, 109–112.

[33] Grosser, D., H.A. Sahraoui, and P. Valtchev(2002): Predicting software stability using Case–Based Reasoning. In *Proceedings of 17th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 295–298.

[34] Mohagheghi, P., R. Conradi, O.M. Killi, and H. Schwarz(2004): An Empirical Study of Software Reuse vs. Defect–Density and Stability. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 282–291.

[35] Hamza, H.S(2004): SODA: A Stability–Oriented Domain Analysis Method. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object–oriented programming systems, languages, and applications*. ACM, 220–221.