http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 7 (3): 474-481, 2008
ISSN 1812-5638

A General Approach for Optimizing Degree of Variability of Software Components

Zhongjie Wang, Dechen Zhan and Xiaofei Xu
Research Center of Intelligent Computing for Enterprises and Services,
School of Computer Science and Technology, Harbin Institute of Technology,
P.O. Box 315, No. 92 West Da Zhi Street, Harbin, Heilongjiang, China

Abstract: In this research, we put forward a general approach for designing components from multiple domain applications by optimizing degree of variability of components, to reach a global optimization between usefulness and usability and maximize global reusability performance of components. A simplified software component model and its usability metrics are firstly introduced. Then process of variability optimization oriented component design is put forward. Three important issues in this approach, i.e., choosing variability policies (negative or positive), determining degree of variability based on semantics abstraction tree (SAT) partitioning and component structure design by composing multiple dimensions, with some primary solutions, are briefly discussed.

Key words: Software components, degree of variability, optimization, usability metrics, semantics abstraction tree, dimensional analysis

INTRODUCTION

One of the most important goals of software engineering is to achieve large-scale reuse. From 1970s there has appeared the concept Software Component (Szyperski, 1998) and researchers imported Reuse-Based Software Engineering (RBSE) (Mili *et al*., 2002) to design software components with higher reusability. There is a core step named Domain Analysis (Kang *et al*., 1990), whose goal is to find commonalities between various applications in a specific domain and encapsulate them in a reusable component by abstraction and variability techniques.

For example, a programmer tries to write a SORT function with an int-type array as its parameter. To improve its reusability, he extends the function so that it could deal with any data types, e.g., float, double, char, etc. Another typical example may be found from Addy *et al*. (1999), where aiming at the domain of waiting queue simulations, a reusable system is required to have the ability of dealing with ten similar applications, e.g., CPU dispatching, self-serve car wash, check-out counters, immigration posts, etc.

Surely domain analysis has a quite good objective and it has been widely put in practice in the past decades. However in real world, there exists an issue that has been little noticed. For instance, we've ever met the following typical scenarios:

- Domain analyzers have designed a good component model but programmers cannot (or are difficult to) transform it to executable codes, just because the component model is so abstract and containing too much variation points that make it quite difficult for programmers to understand and implement.
- Even such components are finally implemented by paying innumerable trials and hardships, some of them are kept on the shelf for long time, even if their reusability is high. This is because a lot of work (e.g., instantiations, configurations or extensions) must be done before reusing such components.

Take philosophy as a metaphor. In philosophy, things in real world are abstracted as two general concepts (substance and spirit) that have the largest reusability, but how many people will directly use them to describe concrete things? -Seldom. Similarly in domain analysis, it is not always true that the higher degree of abstraction, the higher reusability, then the better. Besides reusability, analyzers should carefully consider some other questions, too, e.g., can the abstraction be easily implemented, how much cost must be paid in order

Corresponding Author: Zhongjie Wang, Research Center of Intelligent Computing for Enterprises and Services, School of Computer Science and Technology, Harbin Institute of Technology, P.O. Box 315, No. 92 West Da Zhi Street, Harbin, Heilongjiang, China
Tel: +86-451-86413750 Fax: +86-451-86412664

to reuse it, etc. If implementation cost or reuse cost of a component is too heavy or even severely counteracts the benefits brought from high reusability, then it is not necessary to do so deep abstraction.

Mili *et al.* (2002) decomposed reusability into two sub-metrics, i.e., usefulness and usability. The former one concerns with the extent to which a reusable component will often be needed, while the latter one refers to the extent to which a software component is easy to use. Most of practitioners pay most of their attentions to usefulness while ignore the optimization on usability. In fact, that is a tradeoff between the two metrics and such tradeoff is closely related to the degree of abstraction (or degree of variability) of a component. There has reached a consensus that, the higher degree of variability a component has, the more reuse opportunities it has, then the higher reusability; however at the same time, it will lead to more complex design, lower developing efficiency and higher reuse cost; and vice visa.

Concerning domain analysis and software reuse, there have been a lot of mature theories and methodologies that have been widely applied. For example, there are tens of domain analysis methods, e.g., FOAD (Kang *et al*., 1990), FORM (Kang *et al*., 1998), ODM (Simos, 1995), etc. Various reuse techniques are emphatically imported into Object-Oriented Programming, Component-Based Development (CBD) and Service-Oriented Architecture (SOA). However, all such methods focus on the optimization of usefulness. Although Mili *et al.* (2002) presented the concept usability, they just considered it from extrinsic and intrinsic packaging styles, e.g., rationality of components static structure, completeness and undestandability of components description information, while ignored the tradeoff mentioned earlier.

This research will do some primary attempts on this issue. Reuse-oriented and variability-based component model is firstly analyzed in retrospect, with several usability metrics presented. Then, a general process for component variability optimization is put forward, with some brief discussions on three key technical steps to find an optimal solution for the tradeoff.

SOFTWARE COMPONENT MODEL AND ITS USABILITY METRICS

Three separate parts of a reusable component: The reason a component is reusable is that it contains some functions which are common to various applications. Generally speaking, a reusable may be decomposed into three parts (Mili *et al*., 2002), i.e.,

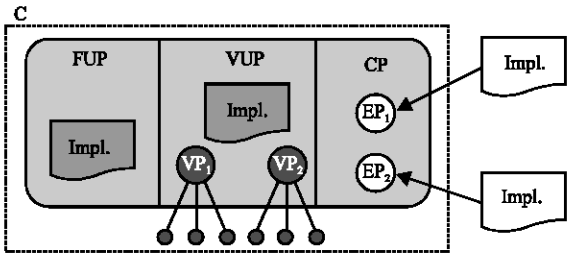

Fig. 1: Three separate parts of a reusable component

- **Fixed Universal Part (FUP):** No matter they are reused in which specific application, such parts are completely the same and could be directly reused without any modifications, extensions or abridgements;
- **Variable Universal Part (VUP):** Functions provided in such parts, abstracted from different but similar applications, have to be instantiated or configured to a concrete form when they are reused in a specific application. To support such purpose, variation point (VP) based mechanism is applied, where each variation point has a set of configuration parameters. Aiming at a specific reuse situation, each parameter is assigned with a specific value so as to make the component exhibit expected behaviors;
- **Customized Part (CP):** Functions in this part are quite different for various applications in a domain and they cannot be abstracted to a unified form. Implementations of this part are in most circumstances not contained in the component but should be further reinforced by application developers when it is reused in a specific application. A component might provide some extension point (EP) based mechanism for such purpose.

Figure 1 shows an example of a reusable component composed of FUP, VUP and CP and there are two variation points VP_1 and VP_2 in VUP and two extension points EP_1 and EP_2 in CP. The dashed rectangle represents scope of the component.

Coarse-grained component reuse: Along with the flourish of reuse-related technologies, one of the most distinct trends in this field is coarse-grained reuse (Helton, 1998), not only for improving reuse efficiency, but also for maintaining a direct mapping from real-life business. Especially in the development of enterprise software and applications (e.g., ERP and SCM), there have gradually appeared business form (e.g., procurement orders, bill of lading, sales order, etc.) based coarse-grained component reuse (Wang *et al*., 2007).

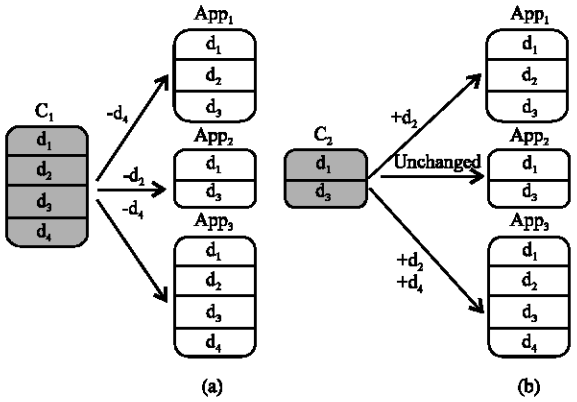

Fig. 2: Negative and positive variability policies (a) Negative variability (NV) and (b) Positive variability (PV)

Design and implementation of such coarse-grained components are more complex and difficult than traditional fine-grained components, not only because it should support multiple complicated application scenarios, but also because unitary dimension cannot fully cover all aspects of its functions but have to make use of multiple dimensions for complete descriptions.

For example, in manufacturing enterprises there are various types of procurement orders (PO), e.g., PO for production materials, PO for fuels, PO for equipments, etc and in order for reuse, we wish to develop one abstract PO component instead of developing three independent PO components. Such abstract component is composed of tens of functional dimensions, e.g., data set, business operations, user interfaces, etc. For the waiting queue simulation system (Mili *et al*., 2002), designers should consider multiple dimensions, too, e.g., topology of service stations, length of service time, customer arrival distribution, dispatching policy, etc.

Negative and positive abstraction: There are two opposite abstraction policies, i.e., positive and negative variability (Mili *et al*., 2002). Positive variability (P/V) starts with a minimal core and selectively adds additional parts based on the presence or absence of features in the configuration models during reuse. While negative variability (N/V) selectively takes away parts of a creative construction model based on the presence or absence of features in the configuration models during reuse (Groher and Voelter, 2007).

For example in Fig. 2, component C_1 is designed following N/V and C_2 is designed following P/V. When they are reused for developing three applications App_1, App_2 and App_3, different modifications should be considered.

In practice, the two policies are both widely applied. But aiming at a specific domain, it should take some time for domain analyzers to elaborately determine which one is more proper to be adopted. Generally speaking, implementing a positive variability is relatively easy, but requires a lot of works when it is reused. Implementing a negative variability is relatively complex, but does not require too much works when it is reused (except pruning those unnecessary functions).

Metrics for usability: In literatures there have numerous metrics for measuring quality and performance of a component (Mili *et al*., 2001). Here we just present several new metrics closely related to usability and variability optimization.

- Usefulness, measured by the total number of applications where a component may be reused, or the percent of applications where a component could be reused in a specific domain. For example, there are n possible applications in a domain and a component may be reused in m ones, then its usefulness is m/n.
- **Development Cost (DC):** Total cost for designing and implementing a reusable component from multiple applications. DC is often closely related to the degree of variability. From our empirical study, developing a high abstract component would pay more cost than, respectively developing n different but similar concrete components, although the latter situation requires developing more LOC than the former. For simplification, we use the average time t for implementing a component c by a programmer who is in average programming level, to measure DC of c.
- **Reuse Cost (RC):** Total cost for reusing a component in a specific application. RC is further considered as the sum of three sub-costs, i.e., cost for deleting those unnecessary functions, cost for instantiating abstract dimensions to required concretions by specifying value of each parameter and cost for implementing unfulfilled functions (i.e., extension).

PROCESS FOR COMPONENT VARIABILITY OPTIMIZATION

Suppose in domain D there are n different applications $\{a_1, a_2, \ldots, a_n\}$. Our purpose is to design one or several components to cover all the n applications as far as possible and reach a tradeoff optimization between usability and usefulness of components.

Firstly we denote applications as the form of multi-dimensional vectors. Suppose that in order to completely describe domain D, m inter-independent dimensions are required, i.e., $\{d_1, d_2, \ldots, d_m\}$, with the following features:

- Some dimensions are common to all applications, while for other dimensions, only one or some of the applications have and the others have not
- Some dimensions are mandatory while others are optional
- It is possible that all applications have the same value on a specific dimension, or have multiple different values
- There are value dependencies between dimensions, e.g., if a dimension d_i is instantiated to a specific value, another dimension d_j must be instantiated to some specific values
- Each dimension depicts one specific semantics aspect of the domain and different dimensions cannot be abstracted as one dimension. Therefore during domain analysis, these dimensions should be separately considered

First we put forward two questions, i.e.,

Q1: Which dimensions should be reserved in final components and which should be ignored? That is to say, choose N/V or P/V policies for each dimension

Q2: If a dimension is to be kept in final components, how about its representation styles? That is to say, does the dimension belong to FUP, VUP or CP? If it belongs to VUP, what degree of variability it should have?

To answer Q1, a method is necessary to help judge the worth of each dimension.

To answer Q2, supposing a dimension d_i has totally t different values $\{v_{i1}, v_{i2}, \ldots, v_{it}\}$ in m applications (of course $t \leq n$ and $m \leq n$) and other n-m applications do not contain d_i at all, we then identify d_i as one of the following five types, i.e.,

S1 : $t = 1, m = n$
S2 : $t = 1, m < n$
S3 : $1 < t \leq n, m = n$
S4 : $1 < t \leq n, m < n$
S5 : $t = 1, m = 1$

Here another two questions are raised:

Q21: For each value of a dimension, should it be reserved in final components or not? That is to say, choose N/V or P/V policies for each value of a dimension;

Q22: For S3 and S4, since there are multiple values for a dimension, what degree of variability of the dimension should be kept in final components? For example, should these values be abstracted to one value with highest degree of variability, to q ($1 < q < t$) values, or just kept the t values un-abstracted?

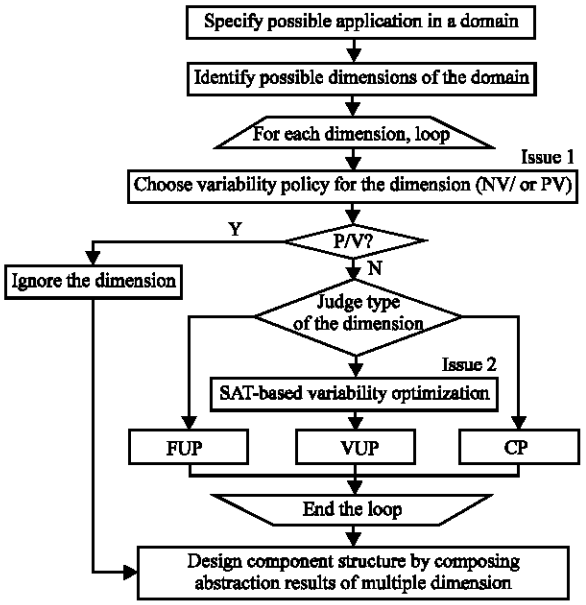

Fig. 3: Process of variability optimization oriented component design

- For Q21, we will adopt to the same method to be discussed
- For Q22, there are three possible solutions
- For S1 and S2, directly form FUP of the final component
- For S3 and S4, use specific strategy to determine the degree of variability, then form one abstract, several semi-abstract or several concrete values to form VUP of the final component
- For S5, directly form CP of the final component

After obtaining FUP, VUP and CP, the last step of component design is to consider how to compose these dimensions together to get the final component.

In conclusion, variability optimization oriented component design process is shown in Fig. 3.

THREE KEY ISSUES ABOUT VARIABILITY OPTIMIZATION

Choosing negative or positive variability policy: As mentioned earlier, there are two separate places where elaborate selection between N/V and P/V is required, i.e.,

- For a specific dimension d_i
- For a specific value v_{ij} of a specific dimension d_i

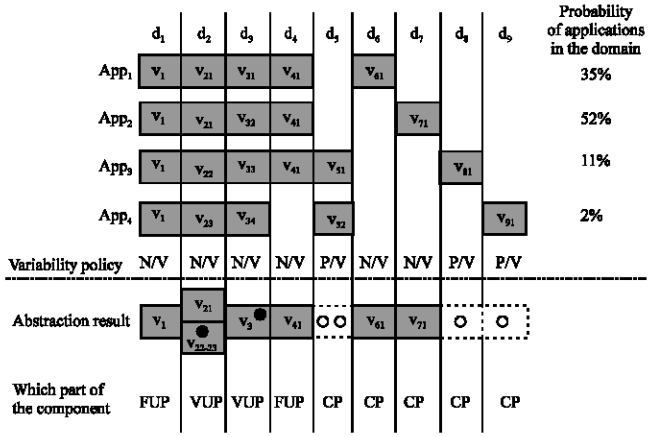

Fig. 4: Strategies for designing FUP, VUP and CP of a component by dimension analysis

Here we apply a very simple strategy to solve this issue, i.e., if a dimension has more chances to appear in most of applications of the domain, then it shows it is valuable to be reused, therefore N/V policy should be adopted; on the contrary, if only a small number of applications contain this dimension, then it has no enough value to be kept in the component and this dimension will be cast away and P/V is adopted.

Formally speaking, suppose the percent of n applications in domain D are, respectively g_1, g_2, …, g_n where

$$\sum_{k=1}^{n} g_k = 1$$

and for a dimension d_i, there are p applications $\{a_{i1}, a_{i2},\ldots, a_{ip}\}$ containing d_i, whose percent are respectively g_{i1}, $g_{i2},\ldots, g_{ip}$. So, if

$$\sum_{k=1}^{p} g_{ik} \geq T$$

(T is a threshold), d_i will be kept using N/V policy, i.e., if such component is to be reused for developing a_{i1}, $a_{i2},\ldots$, a_{ip}, it is not necessary to re-develop d_i; while when it is to be reused for those applications out of $\{a_{i1}, a_{i2},\ldots, a_{ip}\}$, the implementation of d_i has to be manually pruned.

Similarly,

$$\sum_{k=1}^{p} g_{ik} < T$$

if, then d_i will be thrown away by using P/V policy, i.e., if such component is to be reused for developing a_{i1}, $a_{i2},\ldots$, a_{ip}, any extra work are unnecessary for this dimension and for those applications out of $\{a_{i1}, a_{i2},\ldots, a_{ip}\}$, developers have to manually write some code for implementing d_i.

For example in Fig. 4 where there are four applications app_1~app_4 that are described by nine independent dimensions d_1~d_9, we could easily find the best variability policy of each dimension according to the method mentioned earlier.

In real world, it is sometimes difficult to specify probability of each application in the domain before experiencing large-scale or long-period reuse. In this situation, experiences and knowledge of domain specialists, or investigation on requirements of future potential customers, will be imported to help determine whether N/V or P/V is adopted.

When choosing N/V or P/V for each value v_{ij} of a dimension d_i, the same strategy is adopted, i.e., if the total reuse probability of v_{ij} in the domain is above the threshold, it should be implemented and kept in final component by variation point mechanism; otherwise, it will be ignored in final component by extension point mechanism.

As an empirical result, value of the threshold (T) is usually set between 0.25 and 0.3. Practitioners may adjust this threshold according to specific reuse strategies of their own product lines.

Determining degree of variability based on Semantics Abstraction Tree (SAT) partitioning: This sub-section tries to solve the issue that, how to determine degree of variability for optimization.

Because of ideation limitations of human beings, when people think about a complex problem, they usually firstly deal with simpler issues, then gradually with more complex issues and finally the most difficult ones. Therefore when trying to do abstraction on m different values $\{v_{i1}, v_{i2},\ldots, v_{im}\}$ of a specific dimension d_i, steps of the abstraction process usually form a hierarchical tree, which is called Semantics Abstraction Tree (SAT). Figure 5 shows a simple example of SAT.

There are two types of nodes in SAT, i.e., concrete node (CN) and abstract node (AN), where CN are leaf nodes and locate in the bottom layer of SAT and reflect concrete requirements from various specific domain

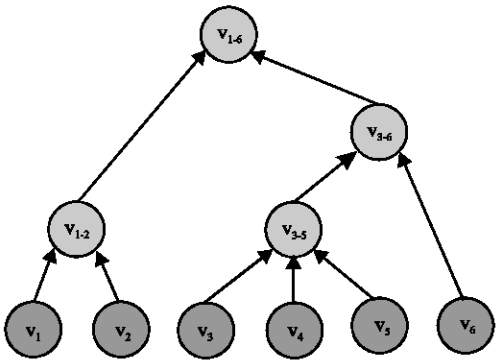

Fig. 5: An example of Semantics Abstraction Tree (SAT)

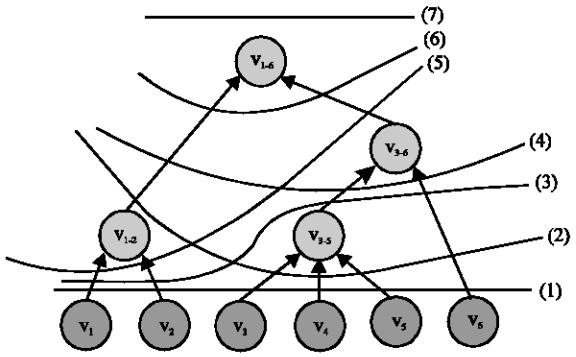

Fig. 6: Partitioning SAT for variability optimization

applications; contrarily, AN are non-leaf nodes and locate in the middle or top layer of SAT and reflect the final or interim abstraction results. There may possibly multiple layers in a SAT and the higher layer a node locates in, the higher degree of variability it has.

In SAT, an edge with the direction from lower to higher layer represents an instantiation-generalization relationship between nodes directly connected by the edge. In OO, CBD and SOA there have been a series of abstraction techniques and Mili *et al.* (2002) did a summarization about these techniques from high level. Due to limited space, we will not list them here in detail.

Take Fig. 5 as examples, where v_1 and v_2 are firstly abstracted to v_{1-2}, v_3, v_4 and v_5 are abstracted to v_{3-5}, then v_{3-5} and v_6 are merged to form v_{3-6} and finally v_{1-2} and v_{3-6} are abstracted to v_{1-6}.

Every node has three attributes related to its reusability, i.e., usefulness, development cost (DC) and reuse cost (RC), all of which has been briefly discussed earlier. For arbitrary node N,

- UF(N), denoting the usefulness of N, is measured by the total number of leaf-nodes contained in the descendant of N
- DC(N), denoting the total cost of abstracting N's all children nodes to N and implementing N as program codes, is measured by the number of Man-Month required to fulfill such abstraction and development
- RC(N), denoting the total cost of reusing N if N is included in the final component, e.g., instantiating N to one of its descendant leaf-nodes, is measured by the number of Man-Month required to fulfill such instantiation and configuration

As mentioned earlier, there is a tradeoff between these attributes. For instance, if we just consider optimization on usefulness, the best solution is to do abstraction as far as possible, i.e., the nearer to the root of SAT, the better. However for optimization on RC and DC, it is completely opposite, i.e., the farther to the root of SAT, the better. Hence it is quite necessary to find a proper boundary to partition SAT as two parts, where we just realize the abstraction under the boundary and for the part above the boundary, we just ignore it, to ensure a global optimization between usefulness, DC and RC.

Aiming at the SAT in Fig. 5, there are seven possible partition solutions, which are marked in Fig. 6 by thick real lines. Take (4) as an example, where it means v_1 and v_2 are abstracted to v_{1-2}, v_3, v_4 and v_5 are abstracted to v_{3-5}, while for the abstractions from v_{3-5} and v_6 to v_{3-6} and from v_{1-2} and v_{3-6} to v_{1-6}, it is not necessary any longer. Therefore the dimension with six concrete values $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ is finally abstracted to three semi-abstract values $\{v_{1-2}, v_{3-5}, v_6\}$.

Since there are multiple partitioning approaches, which one is the best? This is an optimization problem whose objective is represented as the following:

$$\max\left(\sum_{i=1}^{n}(x_i \cdot UF(N_i)) - \sum_{i=1}^{n}(x_i \cdot DC(N_i)) - \sum_{i=1}^{n}(x_i \cdot RC(N_i))\right)$$

Where:

N_i = $(1 \le i \le n)$ is a node in the SAT
x_i = if N_i is above the partition boundary
x_i = 1 if N_i is below the boundary

If size of an SAT is small and there are only finite partitioning solutions, then list all possible solutions and calculate the objective to find the optimal one. If size of an SAT is too large and there are too many partitioning solutions, a linear programming or evolutionary algorithm is employed for reasonable time complexity. Detailed algorithms will not be shown here.

Designing component structure by composing multiple dimensions: From the analysis we have seen that, after

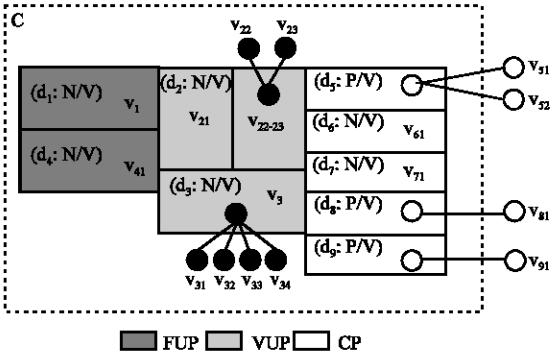

Fig. 7: Composing multiple dimensions into a reusable component

partitioning SAT of a dimension, the number of values of this dimension is reduced to some extent. Besides an extreme situation where the dimension is fully abstracted to a single value, i.e., root node of its SAT (e.g., in Fig. 4, d_3's four values v_{31}, v_{32}, v_{33} and v_{34} are abstracted to one value v_3), in other situations there are still more than one values that should be kept in final components (e.g., in Fig. 4, d_2's three values v_{21}, v_{22} and v_{23} are abstracted to two values v_{21} and v_{22-23}).

In addition, since multiple dimensions are inter-related with each other, we have to find a way to compose the abstraction results of all dimensions together to form an integrated component and fairly identify what structure the component has.

For a dimension d_i, think about five situations (S1 to S5):

- If d_i is satisfied with S1 or S2, d_i will be mapped to a part of FUP
- If d_i is satisfied with S3 or S4, d_i will be mapped to a part of VUP and for d_i's values formed from the partitioning of its SAT, they are mapped to a variation point of the component
- If d_i is satisfied with S5, d_i will be directly mapped to a part of CP
- If d_i is completely ignored or some of d_i's values are ignored by following P/V policy, then add an extension point for each of them in CP

For better understanding, Fig. 7 shows the structure of the final component C according to the abstract results in Fig. 4. FUP of C is composed of two dimensions d_1 and d_4. VUP of C is composed of two dimensions d_2 and d_3, where d_2 is abstracted to two different values v_{21} and v_{22-23} and d_3 is abstracted to a single value v_3. CP of C contains five dimensions d_5, d_6, d_7, d_8 and d_9, where d_6 and d_7 are fully implemented because they follow N/V policy, while d_5, d_8 and d_9 are designed as extension points because they follow P/V policy.

CONCLUSIONS

In this research, we discuss a neglected issue in domain analysis, i.e., tradeoff between usability and usefulness. We elaborately analyze the reason why the tradeoff comes into existence, then briefly put forward our process for designing a reusable component from multiple domain applications. This process seeks a global optimal solution by optimizing degree of variability, containing three key technical problems, i.e., choosing abstraction policy (negative or positive), finding optimal degree of variability and designing structure of the final component. Our work are partially applied in analysis of procurement domain in manufacturing enterprises and statistical shows that DC, RC and usefulness of identified/developed components are synthetically improved.

Future works include: (1) summarize various abstraction techniques and quantitatively measure their development cost and reuse cost; (2) design a more practical method for determining variability policies (P/V or N/V); (3) design and implement a linear programming based algorithm for partitioning SAT.

ACKNOWLEDGMENTS

Research works in this paper are partial supported by the National Natural Science Foundation (NSF) of China (60773064, 60673025), the National High-Tech Research and Development Plan of China (2006AA01Z167) and the Development Program for Outstanding Young Teachers in Harbin Institute of Technology (HITQNJS.2007.033).

REFERENCES

Addy, E., A. Mili and S. Yacoub, 1999. A case study in software reuse. Software Qual. J., 8 (3): 169-195.

Groher, I. and M. Voelter, 2007. Expressing Feature-Based Variability in Structural Models. http://www.voelter.de/data/workshops/MVSPL_GroherVoelter.pdf.

Helton, D., 1998. The impact of large-scale component and framework application development on business. Proceedings of the 3rd International Workshop on Component-Oriented Programming, pp: 163-164.

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-21, Carnegie Mellon University, Software Engineering Institute.

Kang, K.C., S. Kim, J. Lee, K. Kim, E. Shin and M. Huh, 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. Ann. Software Eng., 5 (1): 143-168.

Mili, A., S. Chmiel, R. Gottumukkala and L. Zhang, 2001. Managing software reuse economics: An integrated roi-based model. Ann. Software Eng., 11 (1): 175-218.

Mili, H., A. Mili, S. Yacoub and E. Addy, 2002. Reuse-Based Software Engineering: Techniques, Organization and Controls. John Wiley and Sons Ltd.

Simos, M., 1995. Organization domain modeling (ODM): Formalizing the core domain modeling life cycle. Proceedings of the 1995 Symposium on Software Reusability, pp: 196-205.

Szyperski, C., 1998. Component Software: Beyond Object-Oriented Programming. Addison-Wesley.

Wang, Z.J., D.C. Zhan, X.F. Xu, M.R. Yang and Z. Chen, 2007. Code generator for enterprise software and applications based on business object association model. Comput. Integr. Manuf. Syst., 13 (5): 1021-1029.