Using a 0/1 Knapsack Algorithm for Software Components Selection in Component-based Software System Development

Marjan Kuchaki Rafsanjani[1], Noushin Rakhshan [2]

[1.] Department of Computer Science, Shahid Bahonar University of Kerman, Kerman, Iran
[2.] Science and Research Branch, Islamic Azad University, Kerman, Iran
kuchaki@mail.uk.ac.ir

Abstract: Reusing of the software (SW) components concept started in 1960, when engineering and scientific libraries were used in the SW development to reuse the previously developed functions. This concept is now widely used in SW development as Component Based Development (CBD). CBD is the technology that facilitates the reuse of the existing components into the new ones. One of the most important aspects of Component-Based Software System (CBSS) development is an optimal selection of software components for modules. However, very few researches work on this subject and none of them consider both two important criteria together: cost and cohesion/coupling. In this paper, we have proposed a formulation for profitable components selection for CBSS development. The model has two objectives: maximizing the intra module interactions and minimizing development and adaptation costs between software components and modules with inspiration from a 0/1 knapsack algorithm and this model also consider the modules granularity criterion. This concept in CBD engineering is the complexity of functions that each module in CBSS must do. Granularity criterion in our model is determined with the number of software components that allocate to each module, considering this parameter can help to have same modules in function complexity and run time. This model exploits from a linear formulation that can solve without need to any specific method like genetic algorithm (GA). An example is used to illustrate the proposed methodology.

[Marjan Kuchaki Rafsanjani, Noushin Rakhshan. Using a 0/1 Knapsack Algorithm for Software Components Selection in Component-based Software System Development. Journal of American Science 2011;7(12):641-648]. (ISSN: 1545-1003). http://www.americanscience.org.

Keywords: Software components; Knapsack algorithm; Cohesion; Coupling; Adaptation and development costs

1. Introduction

Reuse of software components concept has been taken from manufacturing industry and civil engineering field. Manufacturing of vehicles from parts and construction of buildings from bricks are the examples. Spare parts of a product should be available in markets to make it successful. Software companies have used the same concept to develop software in parts. Software parts are shipped with the libraries available with SW. These SW parts are called components. Different people have defined the component in different ways. The most popular definition is "Each reusable binary piece of code is called a component". The component concept is similar to object concept of Object Oriented Programming. A component is an independent part of the system having complete functionalities. A component is designed to solve a particular purpose, such as command button and text box of VB. The component is like a pattern that forces the developers to use the predefined procedures and meets the specifications to plug it into the new SW components (Qureshi and Hussain, 2008). For safety critical software a component-based software development (CBSD) approach, using pre-verified library components, can bring savings, particularly in the cost associated with verification. However, a component-based approach is only viable when the overall effort in reusing components is significantly less than the effort in developing (and verifying) the software from scratch (Hemer, 2007). Like other CBSS approaches, there are many challenges that need to be addressed before any real savings can be made (Brereton and Budgen, 2000). These challenges include locating suitable components and adapting them to meet the specific needs of the software engineer (Hemer, 2007). In the context of software engineering, the word reuse often means code reuse, but there is nothing that should prevent the reuse of other software artifacts such as specifications, designs, and test cases. Many researchers argue that greater benefits from software reuse are presumably achieved if higher-level assets such as design artifacts are reused. With current technologies such as object-oriented techniques and domain analysis, software reuse can be amended by concentrating on: reusing artifacts from early stages of the development process rather than later stages, reusing more general solutions rather than specific ones, and reusing more abstract rather than concrete resources. Therefore, current technologies expand the target of reuse to include the reuse of analysis and design artifacts instead of just focusing on program code (Fauzi and Weichang, 2004). In the development of modular-

based conventional software systems, the criteria of minimizing the coupling and maximizing the cohesion of software modules were commonly used. Coupling is about the measure of interactions among software modules while cohesion is about the measure of interactions among the software components which are within a software module. A good software system should possess software modules with high cohesion and low coupling. A highly cohesive module exhibits high reusability and loosely coupled systems enable easy maintenance of a software system (Kwong et al., 2010). Chang and Hua (1994) propose a module-oriented design approach, which is based on a method termed class-function decomposition. This decomposition is not simply an object-oriented decomposition. More meaningfully, it takes a functional view into account. A software environment, called DAGC, is described by Parsa and Bushehrian (2004). The main idea behind the design of DAGC is to facilitate research works in design and development of genetic clustering algorithms for automatic re-modularization of software systems. Within the DAGC environment, clustering algorithms may be assembled or modified by simply selecting the parts from an extendable list of components. Sarkar et al. (2005) have presented a new set of metrics for analyzing the interaction between the modules of a large software system. The metrics are based on the rationale that code partitioning should be based on the principle of similarity of service provided by the different functions encapsulated in a module. Sarkar later presented a new set of metrics that measure the quality of modularization of a non-object-oriented software system (Sarkar et al., 2007). A quantitative approach is proposed to measure and assess the solutions of software modularity based on the criteria of minimal coupling and maximal cohesion (Abreu and Goulão, 2001). None of the previous researches consider both two important criteria together: cost and cohesion/coupling so in this paper we have proposed a formulation that it has considered two factors: maximizing the intra module interactions and minimizing development and adaptation costs between software components and modules. This new model is based on a 0/1 knapsack algorithm and also considers the modules granularity criteria to have same modules in function complexity and run time.

The rest of this paper is organized as follows: the next section gives a background and problem statement. An overview of the model is describes in section 3. Then an illustrative example and a discussion of the results are presented in section 4 and finally conclusions and further works are described in final section.

2. Background and problem statement

Generally, a CBSS is developed based on a top-down approach. Based on the approach, functional/customer requirements are first defined. Then the number and nature of software modules are determined. The next task is to select software components to formulate the modules**.** The domain management stage in CBSS is the planning stage of a reuse program. This planning stage has received little research attention, although it is often cited as being a major barrier to reuse. One of the activities of domain management is domain definition, which consists of defining the scope and boundaries of the envisioned domain. Domain definition must be made consistent with business objectives. The next step is to identify an exhaustive list of candidate components based on customer demands for products (Sundarraj, 2002). So, one of the major problems of CBSS development is how to select software components available in markets to formulate a software module and build a beneficial end-product. There are some different criteria to select a suitable subset of components and a good model should be able to consider a set of more important criteria together. We have considered two factors in our profitable model: maximizing component interaction within each module and minimizing total of adaptation and development costs. For each software component, the organization incurs a development cost, which comprises the programming cost to design and create the software component as well as the maintenance costs. In addition, each end-product has a demand. Finally, for each software component, some modules result directly when the software component gets built, while others can be obtained by an appropriate transformation of the software component, at a certain adaptation cost. On the other hand, the software components should be selected such that the interactions of the software components within a software module are maximized, and interactions of the software components among software modules are minimized. The question that is addressed in this paper is: which set of software components minimizes the total of the development and adaptation costs and simultaneous maximizes the component interaction within each module?

3. Profitable Model

To formulate the problem of selecting software components for CBSS development, the following notations are introduced:

We can calculate the cohesion within the jth module, $(CI_{in})_j$, as follows (Kwong et al., 2010):

$$(CI_{in})_j = \sum_{i=1}^{N-1} \sum_{i'=i+1}^{N} r_{ii'} x_{ij} x_{i'j} \quad (1)$$

Also we compute the total cost for jth module, $(T_{\cos t})_j$, by the following:

$$(T_{\cos t})_j = \sum_{i=1}^{N} c_{ij} x_{ij} + f_i x_{ij} \quad (2)$$

M	**the number of software modules**
N	**the number of software components**
L	**the number of sets of software component**
s_k , $k = 1,\dots,L$	**the kth set of software components**
sc_i, $i = 1,\dots,N$	**the ith software component**
M_j, $j = 1,\dots,M$	**the jth software module**
G_j	**the granularity parameter for module j**
$(T_{\cos t})_j$	**total of development and adaptation costs for module j**
$(CI_{in})_j$	**the number of component interactions within module j (cohesion)**
$r_{ii'}$, $i = 1,\dots,N$	**the number of interactions between** sc_i, sc_j
x_{ij} $i = 1,\dots,N$ $j = 1,\dots,N$	x_{ij} **is a binary variable.** x_{ij} **= 1 if** SC_i **is selected for** M_j**; otherwise,** x_{ij} **= 0**
f_i	**cost of developing** SC_i
c_{ij}	**cost of adapting** SC_i **to module j**
$(profit)_j$	**the quantity sufficiency of module j**
$(weight)_j$	**the number of software components in module j**

By previous section, we want to have two objective functions for each module in our profitable model:

$$Max(CI_{in})_j = \sum_{i=1}^{N-1} \sum_{i'=i+1}^{N} r_{ii'} x_{ij} x_{i'j} \quad (3)$$

$$Min(T_{\cos t})_j = \sum_{i=1}^{N} c_{ij} x_{ij} + f_i x_{ij} \quad (4)$$

The objective function (3) is formulated based on maximizing cohesion within module *j* and the objective function (4) is formulated based on minimizing the total of development and adaptation costs for module *j*. Hence, the profitable problem of software components selection for CBSS development can be considered such as a 0/1 knapsack problem with following objective function:

$$(profit)_j = \max \frac{(CI_{in})_j + 1}{(\cos t)_j} \quad (5)$$

$$(weight)_j \le G_j \quad (6)$$

$$\left\lfloor \frac{L}{M} \right\rfloor \le G_j \le \left\lceil \frac{L}{M} \right\rceil \quad (7)$$

$$G_j \in z, \sum_{j=1}^{M} G_j = L$$

$$\sum_{i \in S_k} \sum_{j=1}^{M} x_{ij} = 1, k = 1,\dots,L \quad (8)$$

$$\sum_{i=1}^{N} x_{ij} \ge 1 \quad (9)$$

$$x_{ij} \in \{0,1\}, i = 1,\dots,N \quad j = 1,\dots,M \quad (10)$$

Where unequal (6) help to have same modules in functions complexity and run time that G_j can estimate in (7) as we described in previous sections. Equation (8) also denotes that only one software component can be selected from a set of alternative software components for a particular module. Unequal (9) is recommended that each module must contain at least one software component.

4. An Illustrative Example and Implications

The example below illustrates the steps involved in applying the profitable model. Consider a situation in which a local software system supplier planned to develop a system, assume that the software system development team of the company made decision that the product can be built using a combination of three modules. A total of 20 software components available in markets make up ten sets of alternative software component ($s_1 - s_{10}$). Only one software component in each set is selected for a particular software module. For example, sc1, sc2 all belong to the set of alternative software components, s1. They can provide similar functions and can be replaced by each other; these are denoted as $sc_1 - sc_{20}$. Table 1 shows the degrees of interaction among the software components. The range of the degrees is 1–10. The degree '1' means a very low degree of interaction while the degree '10' refers to a very high degree of interaction. The degrees of interaction were determined based on the team's judgment in (Kwong et al., 2010). In this case study, development and adaptation costs have been estimated in 0.5–25 range as shown in Table 2. Now the question is that: how we can select a set of software components that minimizes the total of the development and adaptation costs and simultaneous maximizes the component interaction within each module?

Table 1. Interactions among software components (Kwong et al., 2010)

	S_1		S_2			S_3				S_4		S_5	S_6	S_7	S_8	S_9		S_{10}		
	Sc_1	Sc_2	Sc_3	Sc_4	Sc_5	Sc_6	Sc_7	Sc_8	Sc_9	Sc_{10}	Sc_{11}	Sc_{12}	Sc_{13}	Sc_{14}	Sc_{15}	Sc_{16}	Sc_{17}	Sc_{18}	Sc_{19}	Sc_{20}
Sc_1	0	0	5	10	0	0	0	0	4	10	9	0	0	0	0	9	0	0	2	8
Sc_2	0	0	0	8	6	6	2	3	0	10	0	0	0	0	0	0	6	8	4	5
Sc_3	5	0	0	4	0	2	6	3	6	3	0	4	0	10	0	2	0	0	0	0
Sc_4	10	8	4	0	0	9	0	5	6	4	3	0	5	9	5	9	0	8	8	10
Sc_5	0	6	0	0	0	8	0	7	0	4	10	0	0	0	6	8	10	0	2	8
Sc_6	0	6	2	9	8	0	0	0	0	0	0	0	7	0	4	0	8	0	4	3
Sc_7	0	2	6	0	0	0	0	7	0	0	10	0	3	0	7	0	0	0	0	0
Sc_8	0	3	3	5	7	0	7	0	10	8	0	7	0	0	2	4	0	5	0	6
Sc_9	4	0	6	6	0	0	0	10	0	0	0	0	0	0	4	6	3	5	0	5
Sc_{10}	10	10	3	4	4	0	0	8	0	0	5	2	5	0	0	0	4	2	3	8
Sc_{11}	9	0	0	3	10	0	10	0	0	5	0	6	10	6	0	8	4	9	0	0
Sc_{12}	0	0	4	0	0	0	0	7	0	2	6	0	7	0	4	0	5	0	7	0
Sc_{13}	0	0	0	5	0	7	3	0	0	5	10	7	0	7	7	5	9	0	0	4
Sc_{14}	0	0	10	9	0	0	0	0	0	0	6	0	7	0	4	0	2	0	0	0
Sc_{15}	0	0	0	5	6	4	7	2	4	0	0	4	7	4	0	7	0	0	10	0
Sc_{16}	9	0	2	9	8	0	0	4	6	0	8	0	5	0	7	0	0	0	0	7
Sc_{17}	0	6	0	0	10	8	0	0	3	4	4	5	9	2	0	0	0	3	0	0
Sc_{18}	0	8	0	8	0	0	0	5	5	2	9	0	0	0	0	0	3	0	0	6
Sc_{19}	2	4	0	8	2	4	0	0	0	3	0	7	0	0	10	0	0	0	0	2
Sc_{20}	8	5	0	10	8	3	0	6	5	8	0	0	4	0	0	5	0	6	2	0

Some of assumptions must be considered in our profitable model to solve described problem are in follow:

Assumptions

I. We assume that the minimum cost of adapting to module j is 0.5, and maximum cost of adapting to module j is 25, to prevent 0 and ∞ values.

II. When a SC be determined to built in a specific module all SCs that are in the same set of it, be ignored for next steps.

III. For this example G_1 and G_3 are 3 and G_2 is 4.

IV. Until first SC belongs to a module, cohesion parameter values for this module stays 0.

Table 2. Total of development and adaptation costs for module

	M_1	M_2	M_3
SC_1	0.5	3	3.5
SC_2	3.25	2.75	0.5
SC_3	20	15	0.5
SC_4	23	16	2.25
SC_5	19	14	1
SC_6	0.5	1.5	7
SC_7	11	4.25	5
SC_8	7.5	3	2.2
SC_9	0.5	1	5
SC_{10}	5.5	2.3	11
SC_{11}	18	16	0.5
SC_{12}	0.5	0.5	1
SC_{13}	1.2	15	7.3
SC_{14}	0.5	13	16.25
SC_{15}	17	12	5.5
SC_{16}	4	3.25	4.25
SC_{17}	23	11	16
SC_{18}	3.2	0.5	0.5
SC_{19}	17	12.1	11
SC_{20}	20	3	2.5

With these notations and assumptions, a profitable model to select a suitable set of SCs for the above problem can be stated as follows:

In all steps we would have filled tables with calculate below objective function:

$$(profit)_j = \max \frac{(CI_{in})_j + 1}{(\cos t)_j} \qquad (11)$$

For three module M1, M2, M3 considering by values that has been organized in table 1 and table 2.

Step1

	M_1	M_2	M_3
Sc_1	**2**	**0.33**	**0.28**
Sc_2	**0.3**	**0.36**	**2**
Sc_3	**0.05**	**0.06**	**2**
Sc_4	**0.04**	**0.06**	**0.44**
Sc_5	**0.05**	**0.07**	**1**
Sc_6	**2**	**0.66**	**0.14**
Sc_7	**0.09**	**0.23**	**0.2**
Sc_8	**0.13**	**0.33**	**0.45**
Sc_9	**2**	**1**	**0.2**
Sc_{10}	**0.18**	**0.43**	**0.09**
Sc_{11}	**0.05**	**0.06**	**2**
Sc_{12}	**2**	**2**	**1**
Sc_{13}	**0.83**	**0.06**	**0.13**
Sc_{14}	**2**	**0.07**	**0.06**
Sc_{15}	**0.05**	**0.08**	**0.18**
Sc_{16}	**0.25**	**0.3**	**0.23**
Sc_{17}	**0.04**	**0.09**	**0.06**
Sc_{18}	**0.31**	**2**	**2**
Sc_{19}	**0.05**	**0.08**	**0.09**
Sc_{20}	**0.05**	**0.33**	**0.4**

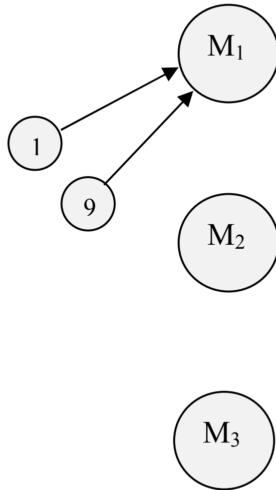
M_1 1 M_2 M_3

yields →

Survivor SCs : $sc_3,\ldots,sc_{20}$

Step2

	M_1	M_2	M_3
Sc_3	**0.29**	**0.06**	**2**
Sc_4	**0.46**	**0.06**	**0.44**
Sc_5	**0.05**	**0.07**	**1**
Sc_6	**1**	**0.66**	**0.14**
Sc_7	**0.08**	**0.23**	**0.2**
Sc_8	**0.12**	**0.33**	**0.45**
Sc_9	**5**	**1**	**0.2**
Sc_{10}	**1.83**	**0.43**	**0.09**
Sc_{11}	**0.54**	**0.06**	**2**
Sc_{12}	**1**	**2**	**1**
Sc_{13}	**0.58**	**0.06**	**0.13**
Sc_{14}	**1**	**0.07**	**0.06**
Sc_{15}	**0.05**	**0.08**	**0.18**
Sc_{16}	**2.25**	**0.3**	**0.23**
Sc_{17}	**0.04**	**0.09**	**0.06**
Sc_{18}	**0.27**	**2**	**2**
Sc_{19}	**0.17**	**0.08**	**0.09**
Sc_{20}	**0.43**	**0.33**	**0.4**

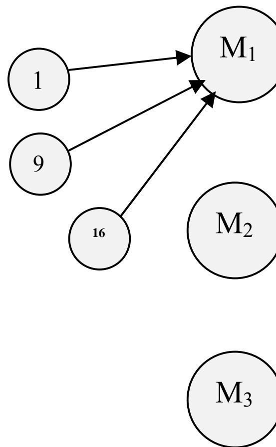


yields →

Survivor SCs : $sc_3, sc_4, sc_5, sc_{10}, sc_{11},...., sc_{20}$

Step3

	M_1	M_2	M_3
Sc_3	**0.76**	**0.06**	**2**
Sc_4	**0.87**	**0.06**	**0.44**
Sc_5	**0.25**	**0.07**	**1**
Sc_{10}	**2.5**	**0.43**	**0.09**
Sc_{11}	**0.73**	**0.06**	**2**
Sc_{12}	**3.33**	**2**	**1**
Sc_{13}	**2.27**	**0.06**	**0.13**
Sc_{14}	**3.33**	**0.07**	**0.06**
Sc_{15}	**0. 5**	**0.08**	**0.18**
Sc_{16}	**4**	**0.3**	**0.23**
Sc_{17}	**0.33**	**0.09**	**0.06**
Sc_{18}	**2.38**	**2**	**2**
Sc_{19}	**0.38**	**0.08**	**0.09**
Sc_{20}	**0.85**	**0.33**	**0.4**



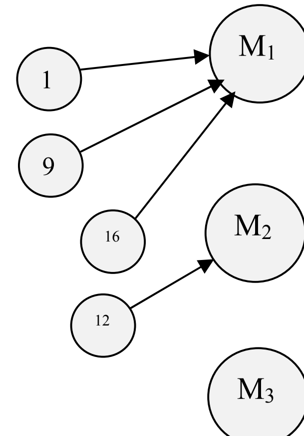
yields →

Survivor SCs :

$sc_3, sc_4, sc_5, sc_{10}, sc_{11}, sc_{12}, sc_{13}, sc_{14}, sc_{15}, sc_{18}, sc_{19}, sc_{20}$

Step4

	M_2	M_3
Sc_3	**0.06**	**2**
Sc_4	**0.06**	**0.44**
Sc_5	**0.07**	**1**
Sc_{10}	**0.43**	**0.09**
Sc_{11}	**0.06**	**2**
Sc_{12}	**2**	**1**
Sc_{13}	**0.06**	**0.13**
Sc_{14}	**0.07**	**0.06**
Sc_{15}	**0.08**	**0.18**
Sc_{18}	**2**	**2**
Sc_{19}	**0.08**	**0.09**
Sc_{20}	**0.33**	**0.4**



yields →

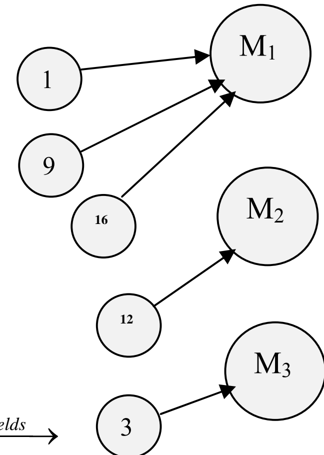
Survivor SCs :

$sc_3, sc_4, sc_5, sc_{10}, sc_{11}, sc_{13}, sc_{14}, sc_{15}, sc_{18}, sc_{19}, sc_{20}$

Step5

	M_2	M_3
Sc_3	**0.32**	**2**
Sc_4	**0.06**	**0.44**
Sc_5	**0.06**	**1**
Sc_{10}	**1.07**	**0.09**
Sc_{11}	**0.42**	**2**
Sc_{13}	**0.51**	**0.13**
Sc_{14}	**0.07**	**0.06**
Sc_{15}	**0.4**	**0.18**
Sc_{18}	**1**	**2**
Sc_{19}	**0.63**	**0.09**
Sc_{20}	**0.28**	**0.4**

yields →



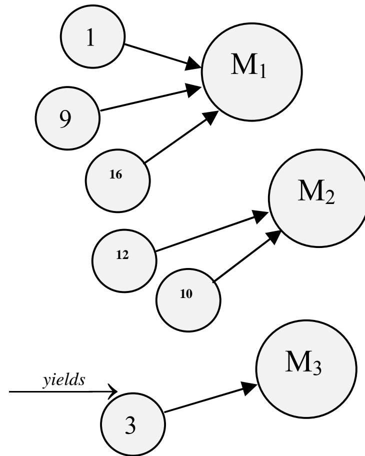
Survivor SCs :

$sc_{10}, sc_{11}, sc_{13}, sc_{14}, sc_{15}, sc_{18}, sc_{19}, sc_{20}$

Step6

	M_2	M_3
Sc_{10}	**1.07**	**0.34**
Sc_{11}	**0.42**	**1**
Sc_{13}	**0.51**	**0.12**
Sc_{14}	**0.07**	**0.71**
Sc_{15}	**0.4**	**0.16**
Sc_{18}	**1**	**1**
Sc_{19}	**0.63**	**0.08**
Sc_{20}	**0.28**	**0.33**

yields

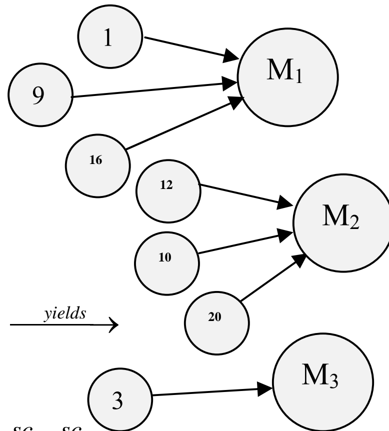


Survivor SCs : $sc_{13}, sc_{14}, sc_{15}, sc_{18}, sc_{19}, sc_{20}$

Step7

	M_2	M_3
Sc_{13}	**0.84**	**0.12**
Sc_{14}	**0.18**	**0.71**
Sc_{15}	**0.47**	**0.16**
Sc_{18}	**1.51**	**1**
Sc_{19}	**0.87**	**0.08**
Sc_{20}	**1.89**	**0.33**

yields

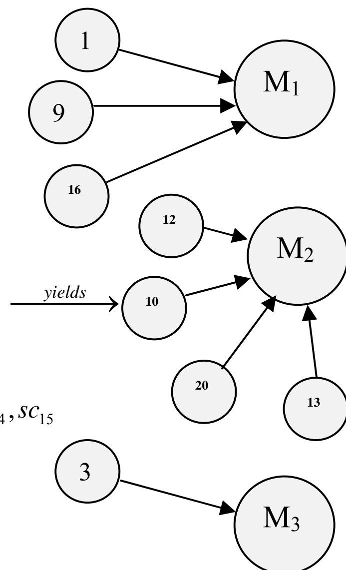


Survivor SCs : $sc_{13}, sc_{14}, sc_{15}$

Step8

	M_2	M_3
Sc_{13}	**1.29**	**0.12**
Sc_{14}	**0.58**	**0.71**
Sc_{15}	**0.84**	**0.16**

yields



Survivor SCs : sc_{14}, sc_{15}

Final Result

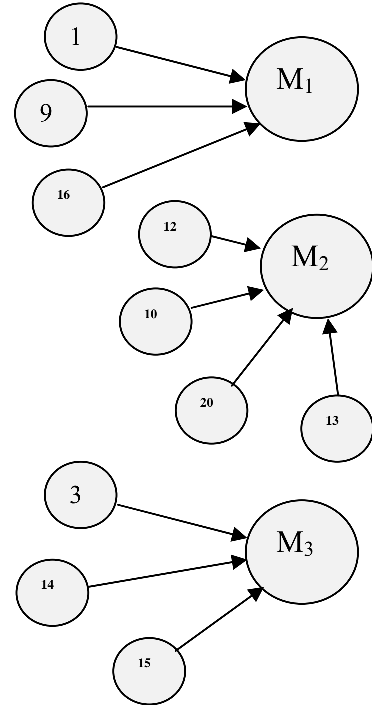


Figure 1 shows the results of solving the problem in question using profitable model. In the figure, the upper chart refers to the interactions between each software component and other software components within its relevant module and the lower chart represents the mean values of interactions between each software component and software components in other modules that it is not belongs them. Table 3 shows the final results of applying 0/1 knapsack model for solving software components selection problem for discussed example. From the figure, it can be noted that for all software components inside dependency is greater than outside dependency and it means that the cohesion and coupling criteria are in a good condition for each modules. Table 4 also shows the binary matrix X for discussed example, from this matrix we can obviously understand that only one software component be selected from each set.

Table 3. Component selection results

G			CI in			Results of SCs selection for modules			Costs		
M_1	M_2	M_3	M_1	M_2	M_3	M_1	M_2	M_3	M_1	M_2	M_3
3	**4**	**3**	**19**	**26**	**14**	**Sc_1,Sc_9,Sc_{16}**	**Sc_{10},Sc_{12},Sc_{13},Sc_{20}**	**Sc_3,Sc_{14},Sc_{15}n**	**5**	**20.8**	**22.25**

Table 4. Showing component selection results in binary matrix X

	Sc_1	Sc_2	Sc_3	Sc_4	Sc_5	Sc_6	Sc_7	Sc_8	Sc_9	Sc_{10}	Sc_{11}	Sc_{12}	Sc_{13}	Sc_{14}	Sc_{15}	Sc_{16}	Sc_{17}	Sc_{18}	Sc_{19}	Sc_{20}
M_1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
M_2	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1
M_3	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

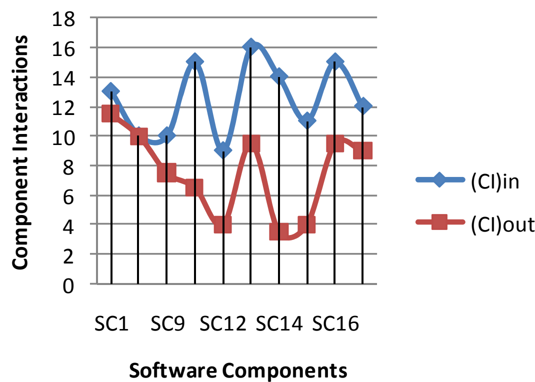


Figure 1. Results of solving the components selection problem using a 0/1 knapsack algorithm

5. Discussion and Conclusion

In this paper, a methodology of selecting software components for CBSS development is proposed. This profitable model with inspiration from a 0/1 knapsack algorithm, minimizes total of development and adaptation costs and maximizes intra module cohesions. This model considers the modules granularity criteria that helps to have same modules in function complexity and run time and in our model we determine it with the number of software components that allocate to each modules and also compared it with the previous studies on CBSS development. So, a modified way of software components selection to consider the total costs and software component interactions simultaneous for CBSS development is introduced. This model exploits from a linear formulation that can solve without need to any specific method like GA. An example of a system design was used to illustrate the proposed methodology. However, this methodology involves some subjective judgments from software development teams, such as the determination of the scores of interaction and the adaptation and development costs.

Corresponding Author:
Marjan Kuchaki Rafsanjani
Department of Computer Science
Shahid Bahonar University of Kerman
Kerman, Iran
E-mail: kuchaki@mail.uk.ac.ir

References

1. Qureshi MRJ, Hussain SA. A reusable software component-based development process model. Advances in Engineering Software 2008:39:88–94.
2. Hemer D. Semi-Automated Component-Based development of formally verified software. Electronic Notes in Theoretical Computer Science 2007:187:173–188.
3. Brereton P, Budgen D. Component-based systems: A classification of issues. IEEE Computer 2000:33:54–62.
4. Fauzi MA, Weichang D. Toward reuse of object-oriented software design models. Information and Software Technology 2004:46: 499–517.
5. Kwong CK, Mu LF, Tang JF, Luo XG. Optimization of software components selection for component-based software system development. Computers & Industrial Engineering 2010:58: 618–624.
6. Chang CK, Hua S. A new approach to module-oriented design of OO Software. Computer software and applications conference 1994:29–34.
7. Parsa S, Bushehrian O. A framework to investigate and evaluate genetic clustering algorithms for automatic modularization of software systems. Lecture Notes in Computer Science 2004:699–702.
8. Sarkar S, Kak AC, Nagaraja NS. Metrics for analyzing module interactions in large software systems. 12th Asia Pacific software engineering conference 2005:264–271.
9. Sarkar S, Rama GM, Kak AC. API-based and information theoretic metrics for measuring the quality of software modularization. IEEE Transactions on Software Engineering 2007:33(1):14–32.
10. Abreu FB, Goulão M. Coupling and cohesion as modularization drivers: Are we being over-persuaded. Fifth European conference on software maintenance and reengineering. Washington, DC, USA: IEEE Computer Society 2001.
11. Sundarraj RP. An optimization approach to plan for reusable software components. European Journal of Operational Research 2002:142:128–137.

11/21/2011