Optimal Component Selection for Component Based Software Development using Pliability Metric

Jeetendra Pande
Uttarakhand Open University,
Haldwani, 263139,INDIA
Research Scholar, UTU,
Dehradun
(+91) 99270-50094

jpande@uou.ac.in

Christopher J. Garcia
College of Business
University of Mary
Washington
Fredericksburg, Virginia
22401
USA
(540) 654-1456

cgarcia@umw.edu

Durgesh Pant
School of CS & IT
Uttarakhand Open University
Haldwani, Uttarakhand
263139
INDIA
(+91) 05946-261122

dpant@uou.ac.in

ABSTRACT

Component Based Software Development (CBSD) has gained widespread acceptance as it often results in higher quality software with a significant reduction in development time and costs. A key idea behind CBSD is the extensive reuse and composition of pre-existing modules into new software. In this paper we introduce the *pliability* metric, which is well suited to a component-based orientation and extends previous metrics. Pliability is a flexible measure that assesses software quality across different quality attributes in terms of the quality of its components. In addition, we have developed an optimal component selection model based on integer programming, for maximizing pliability. Through computational experimentation we demonstrate that this model is capable of finding optimal solutions to problems with a very large number of components and requirements in a short time.

Categories and Subject Descriptors

D.2.8 [**Software**]: Metrics – *Process metrics*

General Terms

Measurement, Design

Keywords

Component Based Software Development (CBSD), Componentoff-the-shelf (COTS), Pliability, Component Based System (CBS).

1. INTRODUCTION

The size of large software projects often exceeds 10 million lines of code. The development time for such large projects may approach a decade if traditional software development approaches are followed. Such a timeframe is most often not feasible as technology changes rapidly and requirements seldom remain stationary for such a long period of time. To cope with this complexity, the Component-Based Software Development (CBSD) paradigm has gained widespread usage, largely out of necessity. In traditional approaches the code is developed from scratch or selected from in-house libraries. CBSD, however, involves the purchase and integration of pre-made software components and makes extensive use of third-party components. This represents an evolution in software engineering practice similar to the paradigm shift in electronics engineering that took place several decades ago, when electronics began to be designed and manufactured from pre-made components purchased from specialized component manufacturers. As a result, similar benefits are being realized within software engineering including the ability to produce much larger and more complex systems not previously possible, reduced time-to-market, increased interoperability, larger return on software development, significantly better reliability, and ease of quality certification.

Traditional software product and process metrics have had mixed success in measuring the utility of software components, which is necessary for quality and productivity improvement within organizations adopting component-based software estimation. Thus, there is a pressing need for development of a new component-based evaluation metric. One of CBSD's aims is enhancement of software quality, which can be achieved by improving maintainability, functionality, security, cost and other quality aspects. One of the critical considerations is that many different quality attributes must be considered and traded off in the final system. Moreover, the relative importance of each of these is not static but depends on the type of software system being developed. As a result, a flexible evaluation approach is needed in order to enable optimal component selection based on the importance of different quality metrics with respect to the system being developed.

Because modern software systems are complex and may involve large numbers of components and requirements, a component selection approach is needed that is also capable of addressing large-scale problems. In this paper we develop the pliability metric, a flexible metric for evaluating software quality in a manner tailored to the needs of the problem at hand. The pliability metric is particularly suited to the component-oriented approach and measures software quality as a function of the quality of individual components. We also develop an integer programming-based optimization model for selecting components to maximize the pliability achieved. This model is based on the set-covering problem and is highly tractable in practice. Although strictly speaking set covering is NP-hard, in practice the linear relaxation is typically an integer solution or very close to one [2]. As a consequence, integer programming is typically a very effective solution method for problems with set covering constraints. We demonstrate the capability of our model to solve very large problems quickly using a set of diverse benchmark problems.

This paper's organization is as follows: Section 2 provides an overview of the literature, Section 3 defines the problem, Section 4 defines the *Pliability* metric and describes our model for component selection based on integer programming, Section 5 describes the design of the computational experiments used to validate the model, Section 6 covers the experiment results and discussion while section 7 lists the conclusions and future scope.

2. LITRATURE REVIEW

Many evaluation and selection techniques have been suggested by different researchers. Kontio[8] proposed the Off-The-Shelf Option(OTSO) approach for component selection and later proposed a modified version of OTSO[9]. The OTSO approach for component selection is based on progressive filtering. This approach used two evaluation criteria for COTS comparison, first the value of the COTS products and second the cost of the COTS product. Lichota, Vesprini, and Swanson[12] proposed a generic component architecture better known as the PRISM (Portable, Reusable, Integrated, Software Modules) approach, which can be used for integrating software products as system components. PRISM highlighted the significance of prototyping, both outside and inside the context. This technique also addresses the build-versus-buy decision with respect to integration within the architecture.

Maiden and Ncube[13] proposedthe PORE (Procurement-Oriented Requirements Engineering) approach for component selection, which suggests that the requirements should be elicited and analyzed at the same time when the COTS products are evaluated. Chung[11] included Non-Functional Requirements while making judgments on COTS components with similar functionality. Ochs, Pfahl, Chrobok-Diening, and Nothhelfer-Kolb[15] emphasized experts' knowledge and suggested the COTS Acquisition Process (CAP), which is based on utilizing experts' knowledge for a customizable evaluation process. Comella-Dorda, Dean, Morris, and Oberndorf[4] proposed PECA (Plan, Establish, Collect and Analyze), which provides the guidelines for a tailorable COTS selection process. Gregor, Hutson, and Oresky[6] proposed the Storyboard approach, which helps the customer to understand their requirements better. Use of use-case and screen-capture during the requirements engineering phase improves understanding of the requirements by the customer. This facilitates the customer in selecting appropriate COTS products as the requirements are better understood. Burgues, Estay, Franch, Pastor, and Quer[3] proposed a combined selection approach for COTS components based on the distinction of local and global levels.

At the local level all the individual selection processes of different business areas are located under the supervision of the process at the global level. At the global level the combined selection process takes place, which results in selection of the best combination of COTS. Boehm, Port, and Yang[1] proposed the three step Win-Win Spiral Model, which uses a risk-driven approach to first identify risk, then to analyze risk and finally resolve the risk in an iterative evaluation process. Grau, Carvallo, Franch, and Quer[5] proposed the DesCOTS system (Description, evaluation and selection of COTS components), which is also centered on component selection from a risk-management perspective. Mohamed, Ruhe, and Eberlein[14] proposed the Mismatch Handling for COTS Selection (MiHOS), which addresses COTS mismatch between system requirements and COTS products during and after the selection process. Kwong, Mu, Tang, and Luo[10] introduced an approach for component selection that aims to maximize functional performance and cohesion. This approach represents a major shift in paradigm toward an optimization orientation by using genetic algorithm.

Despite the many proposed approaches, selection of COTS products is largely carried out using ad hoc methods[16]. The primary reason for this is that the evaluation and selection methods are too complex and labor intensive for CBS developers to understand and implement. In addition, these methods are generally unable to scale up for use with larger systems. With the exception of the methodology proposed by Kwong et. al.[10], use of optimization has been largely ignored, despite higher levels of automation and better quality of solutions that it can offer, particularly for large systems. Except PRISM none of the techniques addressed the build-versus-buy issue, which is very crucial for CBSD. In this paper we introduce a component evaluation and selection approach intended to address these issues. Our component evaluation approach utilizes multiple dimensions of quality and can incorporate virtually any quality measure, enabling flexibility and tailoring to the type of system being designed. Components may be evaluated independently and in parallel, which allows the evaluation to be scaled up as necessary. In addition, we employ optimization for selecting the best set of components to achieve the system requirements based on user-defined quality parameters. While Kwong el.al.[10] use a genetic algorithm (which is an approximate method) to optimize functional performance and cohesion, we use integer programming (which is an exact method) to optimize a more general and customizable set of quality attributes. We demonstrate the capability of this method to solve very large problems quickly through computational experiments.

3. PROBLEM DEFINITION

The problem addressed in this paper may be formally stated as follows: We are given a set of requirements SR and a set of components SC. Each component $i \in SC_i$ covers (i.e. satisfies) a subset of requirements $R(i) \in SR$.

In addition, we are given a set of software quality attributes A (such as performance, scalability, reliability and others), where each h ∈ A has a certain level of relative importance expressed by a weighted value. We need to select a subset $C \subseteq SC$ that collectively covers all requirements in SR in a manner that maximizes the weighted quality attribute levels achieved while balancing against costs. In essence, this problem may be viewed as a benefit minus cost maximization problem.

4. COMPONENT SELECTION FRAMEWORK

4.1 Software Quality Evaluation

For the development of a software system, we may be concerned with any number of quality attributes. We may define the overall system quality measure as Q, based on a set of quality attributes A as proposed by Voas and Agresti[18]. The set A includes reliability, performance, fault tolerance, safety, security, availability, testability, and maintainability. Other measureable quality attributes deemed important may also be included in A in a straightforward manner. According to this scheme, the overall software quality Q may be understood as weighted linear combination of the values for each of these attributes:

$$Q = w_R R + w_P P + w_F F + w_{Sa} Sa + w_{Se} Se + w_{Av} Av + w_T T + w_M M \qquad (1)$$

Where ,
R = Reliability
P = Performance
F = Fault tolerance
Sa = Safety
Se = Security
Av = Availability
T = Testability
M = Maintainability

This approach provides a robust perspective on software quality that considers a full spectrum of quality concerns. The values $w_R, w_P, w_F, w_{Sa}, w_{Se}, w_A$, and w_T denote the weights assigned to the corresponding quality attribute. The sum of all the weights assigned to all the quality attributes is equal to 1 i.e.

$$w_R + w_P + w_F + w_{Sa} + w_{Se} + w_A + w_T = 1 \qquad (2)$$

This approach facilitates a simple, flexible, and consistent way to evaluate and compare the total software quality of proposed designs based on the needs of the stakeholders. The weighting for each attribute depends on the type of software. For a financial system, the weight for security would probably be higher than that for safety while the weight for testability may be less. For a safety-critical system, the key attributes would probably be reliability, performance, safety, fault tolerance, and availability. For an e-commerce system, the key weighted attributes would be reliability, performance, availability, security, and maintainability.

The approach proposed by Voas and Agresti[18] may be extended for use within a component-oriented context if we understand each candidate component to contribute a certain amount of value toward each individual quality attribute. However, each quality attribute has its own unit of measurement. Different types of measurement units are typically not able to be compared in a meaningful way. In order for the weighting scheme described above to make meaningful comparisons it is necessary for all quality metrics to be of equal scale. For example, a Maintainability score of 5.5 and a Reliability score of 5.5 should both contribute equally toward the overall software quality if their weights are equal. Given the diverse measurement schemes that may be employed it is necessary to normalize all quality attribute measures for the components so they may be compared appropriately. In order to do this we first define the following:

q_{hi} = The normalized level of quality attribute $h \in A$ contributed by component i

To enable appropriate comparability, we use a 0-10 normalized scale for all quality attributes. We may compute the normalized quality attribute level for any component i with respect to any quality attribute h by using the ratio of the raw measured value of i with respect to h (in whatever measurement is appropriate for h) to the maximum raw measured value of h attained by any available component. Thus, we let Max(h) denote the highest raw value achieved by any component for quality attribute h (in the measurement units used for the attribute h). We then assign the normalized level of each component i for quality attribute h as follows:

$$q_{hi} = \left(\frac{RawQA(h,i)}{Max(h)}\right) \times 10 \qquad (3)$$

Using this scheme, quality measures are normalized in a way that components may be meaningfully compared according to the weights assigned by the stakeholders.

4.2 Pliability Metric and Optimal Component Selection

The net value of any investment is always understood as the total benefit minus the total cost. The pliability metric used to determine an optimal component selection is based on this principle. The pliability of a system is defined as the total weighted, normalized quality minus the total normalized cost. For reasons that become apparent when we discuss the component selection model, it is necessary to normalize costs in a manner different from quality levels. Thus, we withhold discussion of the cost evaluation and normalization until after presenting the component selection model below. To define pliability in a precise manner, we first define the following:

c_i = Normalized cost of component i

x_i = 1 if component i is selected, 0 otherwise

Consequently, the software system pliability P is defined as follows:

$$P = \sum_{h\in A}\sum_{i\in SC} w_h q_{hi}\, x_i - \sum_{i\in SC} c_i x_i \qquad (4)$$

Here, $\sum_{h\in A}\sum_{i\in SC} w_h q_{hi}\, x_i$ is the total weighted normalized quality achieved by the components selected, and $\sum_{i\in SC} c_i x_i$ is the total normalized costs that result from this selection. Our objective is to select a set of components from SC that covers all requirements in SR and maximizes the total pliability. We devise an optimization model based on integer programming, which uses pliability as the criteria for component selection. Because this problem involves selecting components to cover a set of requirements there are clear linkages to the classical set covering problem as formulated by Toregas et. al. [17]. Although strictly speaking this problem is NP-hard, in practice integer programming is typically a very effective solution method for problems with set covering constraints [2].

To formulate the model we first restate the parameters and decision variables defined above. We also introduce the parameter b_{ij}.

Parameters:

A = The set of system quality attributes we are interested in

w_h = The relative weight collectively assigned by stakeholders to quality attribute $h \in A$

c_i = Normalized cost of component i$\in SC$

b_{ij} = 1 if component i satisfies functional requirement j$\in SR$, 0 otherwise

Decision Variables:

x_i = 1 if component i is selected, 0 otherwise

Model:

$$Maximize\ z = \sum_{h\in A}\sum_{i\in SC} w_h q_{hi}\, x_i - \sum_{i\in SC} c_i x_i \qquad (5)$$

Subject To:

$$\sum_{i\in SC} b_{ij} x_i \geq 1 \qquad \forall\, j \in SR \qquad (6)$$

$$x_i \in \{0,1\} \qquad \forall\, i \in SC \qquad (7)$$

The objective function in equation (5) is simply pliability as defined in eq. (4). Constraints in equation (6) are the set covering constraints that ensure all functional requirements in SR are met by our selected set of components. Finally, constraints in eq. (7) ensure binary variable domain.

4.3 Cost Evaluation

Many costs are involved in attaining and integrating a component into a software system. Kaur & Mann[7] provide a detailed accounting of the composite costs involved in incorporating a new component into a software system. These include costs associated with searching for and assessing the component, the component purchase cost, costs associated with integrating the component, and a number of sundry capital costs including maintenance, training, and others. Thus, the component selection cost model of [7] may be used to estimate the raw cost of selecting and incorporating a component into the software system.

The pliability metric defined above relies on normalized quality levels and costs. Based on the pliability equation (4), we may compute the net benefit NB_i of component i as follows:

$$NB_i = \sum_{h \in A} w_h q_{hi} \, x_i - c_i x_i \qquad (8)$$

We may thus express pliability as the sum of net benefits achieved by our selected components:

$$P = \sum_{i \in SC} NB_i \qquad (9)$$

When viewed in this manner it becomes immediately obvious that if we wish to maximize pliability then we should always select *all* components with positive net benefit. In cases where all candidate components have positive net benefit this would lead to selection of all components – which will under most circumstances provide overly redundant functional coverage and result in grossly suboptimal solutions. The reason for this is because unneeded functional coverage is rewarded rather than penalized when net benefits are positive. We can correct this by ensuring all net benefit quantities are negative. Thus, components with higher net benefit are negative but closer to zero, while those with lower net benefit have larger negative values. This provides the basis for the cost normalization method.

To normalize our costs we use a scale ranging from 10-20, where values closer to 10 denote lower costs while those closer to 20 denote higher costs. Because our normalized quality scale ranges from 0-10, this cost scale ensures all net benefits are negative as discussed above. We let C_{max} denote the highest raw (monetary) cost found among the components in SC. Additionally, we let $\text{RawCost}(i)$ denote the raw monetary cost of adding component i as described above [7]. We may then assign each component $i \in SC$ its normalized cost c_i as follows:

$$c_i = 10 + \left(\frac{\text{RawCost}(i)}{C_{max}} \right) \times 10 \qquad (10)$$

By our quality normalization scheme defined in eq. (3) it is impossible for any component to achieve a normalized quality level greater than 10. As discussed surrounding eq. (4), the total weighted normalized quality achieved for component i is $\sum_{h \in A} w_h q_{hi} \, x_i$. Since the quality attribute weights sum to 1, it follows that this total weighted normalized quality achieved for any component i cannot be greater than 10. Thus, the normalization scheme of (10) will always result in a negative net benefit as defined in (8) and facilitate optimal component selection.

4.4 Incomplete Functional Coverage and Buy vs. Build Decisions

The model discussed up to this point has assumed that all functional requirements can be covered by at least one of the available components. In reality this may not be the case, and in such a scenario the solution is simply to develop the functionality in-house. Alternatively, the capability to build any number of functions in-house may exist, even though these may also be able to be adequately covered by selecting appropriate existing components. In this case a firm is faced with a complex buy versus build decision in which the building versus buying of a particular capability may affect other functional requirements.

The model that we have introduced may handle both cases in a relatively straightforward way. We may express each functional requirement j that we are capable of developing in-house as an "artificial" component to be included in the available set of components. This artificial component covers exactly one requirement j and the cost and quality level achieved for each quality attribute is estimated. In addition, we may be able to develop the function with differing levels of quality at different costs. In such a case multiple artificial components may be included in the component set, each with their respective costs and quality attribute levels. Using this approach we are thus able to incorporate functional coverage as well as complex buy versus build decisions in a relatively straightforward manner.

5. COMPUTATIONAL STUDY DESIGN

In light of today's highly complex software systems, a component-selection approach must be capable of taking large numbers of components and functional requirements into account in order to have practical value. In order to demonstrate the computational tractability and practical utility of our model we performed a computational experiment using a generated set of realistically-large benchmark problems having varying numbers of components and requirements. We implemented our model using ILOG OPL and used the CPLEX solver version 12.2 to solve each problem. All experimentation was performed on a 2.6 GHz computer with 4 GB of memory running Windows 7.

Our computational experiment has two objectives. The first is to understand the magnitude of time that may be expected to obtain solutions for problems of different sizes. The second is to understand the impact of the number of components, the number of requirements, and their interaction on expected solution time. We thus employ a 2^2 factorial experiment design with two problem generation parameters: the number of components and the number of requirements. For each parameter we utilized two levels, low (-) and high (+), as shown in Table 1.

Table 1. Problem generation parameter values

Parameter	Low (-)	High (+)
Components	200	500
Requirements	500	1000

This design results in four parameter combinations: low/low, low/high, high/low, and high/high. For each of these combinations we generated ten replicate problems, giving a total of 40 problems instances in all. Problems were generated based on the parameters in Table 1 using a computer program. Within each problem, each requirement was covered by between 2% and 98% of the components available. This coverage percent for each requirement was generated using a uniform distribution over [0.02, 0.98]. The normalized cost for each component was generated using a uniform distribution over (0, 10]. The normalized benefit q_{hi} for each component i with respect to QA metric h was also generated using a uniform distribution over (0, 10].

In each benchmark problem, the weights used for each QA metric were randomly generated to sum to 1. To generate these weights, each QA metric h was assigned a random number r_h in [0.5, 1]. The raw total T of these was then summed:

$$T = \sum_{h \in A} r_h \qquad (11)$$

The final weight w_h of each QA metric h is calculated as:

$$w_h = \frac{r_h}{T} \qquad (12)$$

Thus, this scheme enables random normalized weights to be generated in a manner that guarantees that they sum to 1.

6. RESULTS AND DISCUSSION

The first objective of the computational experiment was to gauge the expected solution time for different-sized problems. The average, minimum, and maximum times required for optimal solutions to be found for the problems within each parameter combination are shown in Table 2.

Table 2. Required solution times for each problem type

Combination		Solution Time (seconds)		
Components	**Requirements**	**Average**	**Min**	**Max**
-	-	0.47	0.28	0.75
-	+	1.14	0.39	2.86
+	-	3.51	1.03	7.99
+	+	18.45	11	35.6

By examining Table 2, it is clear that this model may be used to solve problems with very large numbers of components and requirements. In particular, optimal solutions for the largest problems instances (which contain 500 components and 1000 requirements) were obtained in less than forty seconds.

The second objective of the experiment was to understand how different numbers of components, requirements, and their interactions affect the solution time. Table 3 shows an ANOVA for the different problems.

Table 3. ANOVA results for solution times (* denotes significant at alpha = 0.001)**

Factor	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
Comp.	1	1034.34	1034.84	60.98	2.94E-09	***
Req.	1	610.34	610.34	35.967	7.01E-07	***
Comp. * Req.	1	509.36	509.36	30.02	3.46E-06	***
Residuals	36	610.92	16.97			

By examining Table 3 it may be seen that both main effects (number of components and number of requirements) have a significant impact on required solution time. Thus, as either one of these increases the required solution time increases as well. In addition, the interaction effect was also statistically significant and indicates a "synergy" between the two factors with regard to impact on solution time. This may be seen in Table 2, where the high/high combination required significantly more time than the sum of the high/low and low/high combinations.

In summary, this model appears to be highly tractable and useful on large, realistic problems. As components and requirements increase there is a relatively proportional (i.e. within the same magnitude) but significant increase in required solution time.

7. DISCUSSION AND CONCLUSION

This work answers the general question of how to address the issue of component selection based on component costs and quality dimensions. A model based on integer programming is proposed, the goal of which is to maximize the pliability of the overall system by designing a metric called the *Pliability* metric, which is helpful in component selection. This new metric provides a comprehensive approach for evaluating software quality while enabling stakeholders to define and prioritize the quality attributes deemed important. The pliability metric was developed to support a component-oriented approach to software development and considers the end software quality achieved in terms of the quality of the components selected.

In order to enable optimal component selection we developed an optimization model based on integer programming and implemented it using ILOG OPL and CPLEX solver. This model utilizes an objective function that maximizes the pliability attained, and we showed how costs and benefits may be normalized in order to ensure appropriate comparison. We conducted computational experimentation to determine the magnitude of time expected to be required for solving problems of various sizes and also to determine the effect of different numbers of components and requirements on the solution time. We found this model to be highly effective and able to solve even very large problems optimally (with 500 components to choose from and 1000 requirements to cover) in considerably less than one minute. This model can clearly support large and highly complex industrial software development efforts. The outcomes of the experimentation have been encouraging and the model appears to be highly tractable and useful on large, realistic problems.

8. FUTURE SCOPE

The field of quality attribute determination of component-based system is extensive and more research should be performed in this field. Future work in the development of component-based technologies could include determination of more quality metrics for components that are easy to calculate and more feasible to use. One final and important problem to be investigated is how to devise a formal methodology for determining the relative weights to be assigned to the different quality metrics based on stakeholder input. In this regard, a potential approach may be to use the analytic hierarchy process. However, further investigation is needed.

9. REFERENCES

[1] Boehm, B., Port, D. and Yang, Y. 2003. Winwin spiral approach to developing cots-based applications. In *Proceedings of 5th International Workshop on Economic-Driven Software Engineering Research,* page 57.

[2] Bramel, J. and Simchi-Levi, D. 1997. On the effectiveness of set covering formulations for the vehicle routing problem with time windows. *Oper. Res.*,295–301.

[3] Burgu´es, X., Estay, C., Franch, X., Pastor, J.A. and Quer, C. 2002. Combined selection of cots components. In *Proceedings of 1st International Conference on COTS-Based Software Systems,* Springer-Verlag Lecture Notes in Computer Science, 54–56.

[4] Comella-Dorda, S., Dean, J.C., Morris, E.J. and Oberndorf, P.A. 2002. A process for cots software product evaluation. In *Proceedings of 1st International Conference on COTS-Based Software Systems,* Springer-Verlag Lecture Notes in Computer Science, 86–96.

[5] Grau, G., Carvallo, J.P., Franch, X. and Quer, C. 2004. Descots: a software system for selecting cots components. In *Proceeding of Euromicro Conference-2004.*(Sep. 2004),118–126

[6] Gregor, S., Hutson, J. and Oresky, C. 2002.Storyboard process to assist in requirements verification and adaptation to capabilities inherent in cots. In *Proceedings of 1st International Conference on COTS-Based Software Systems*, Springer-Verlag Lecture Notes in Computer Science, 132–141.

[7] Kaur, A. and Mann, K.S. 2010. Component selection for component based software engineering. *International Journal of Computer Applications*, 2, 1(2010),109–114.

[8] Kontio, J.1995. *Otso: A systematic process for reusable software component selection,* Technical Report. University of Maryland.

[9] Kontio, J., Caldiera, G. and Basili. V. R.1996. Defining factors, goals and criteria for reusable component evaluation. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*,(Toronto, Canada, November 12-14, 1996), CASCON '96, IBM Press, 21–24.

[10] Kwong, C.K., Mu, L.F., Tang, J.F. and Luo, X.G.2010. Optimization of software components selection for component-based software system development. *Comput. Ind. Eng.*, 58,4.(May 2010). 618 – 624. DOI=10.1016/j.cie.2010.01.003 http://dx.doi.org/10.1016/j.cie.2010.01.003

[11] Chung, L., Nixon, B.A., Yu, E. and Mylopoulos, J. 2000. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishing,.

[12] Lichota, R.W. Vesprini, R.L. and Swanson, B. 1997. Prism product examination process for component based development. In *Proceedings of Fifth International Symposium on Assessment of Software Tools and Technologies* (Pittsburgh, Pennsylvania, USA , June 2-5, 1997). SAST '97.IEEE Computer Society, 61–69. DOI= http://doi.ieeecomputersociety.org/10.1109/AST.1997.599912

[13] Maiden, N.A. and Ncube, C. 1998. Acquiring cots software selection requirements. *IEEE Softw.*15,2.(April 1998), 46–56.

[14] Mohamed, A., Ruhe, G. and Eberlein, A. 2007. Decision support for handling mismatches between cots products and system requirements. In *Proceedings of the Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems* (Washington DC, USA, 2007). ICCBSS '07, 63–72, DOI=

http:// 10.1109/ICCBSS.2007.13

[15] Ochs, M., Pfahl, D., Chrobok-Diening, G. and Nothhelfer-Kolb, B. 2000. A cots acquisition process: Definition and application experience. ISERN Report.

[16] Ruhe, G. 2003. Intelligent support for selection of cots products. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, Springer-Verlag, U.K., 34–45.

[17] Toregas, C., Swain, R., ReVelle, C. and Bergman, L. 1971. The location of emergency service facilities. *Oper. Res.*, 19,6(1971). 1363–1373, 1971. DOI=

http:// 10.1287/opre.19.6.1363

[18] Voas, J. and Agresti, W. W. 2004. Software quality from a behavioral perspective. *IT Professional*, 6,4 (July 2004), 46–50.