International Journal of Computer Applications (0975 – 8887)
Volume 35– No.8, December 2011

Precision in Rapid Application Development and Reusability of Software Components for Greater Performance using Ranking Mechanism

P.K. Suri
Dean, Research and Development; Chairman, CSE/IT/MCA, HCTM Technical Campus, Kaithal, Haryana, India.

Sandeep Kumar
Assistant Professor and Head, Faculty of Computer Appl., Galaxy Global Group of Institutions, Ambala, Haryana, India

Gurdev Singh
Lead Engineer, Samsung Electronics,
Noida, UP, India

ABSTRACT
Software component is a cohesive software module that contains the semantically related functionality. The term "use component anywhere" is not that true as it seems to be. The important thing is that for using the components, there should be a well-defined framework where they will be used. The convention is exactly the same as that of IC (integrated chip) socket on a circuit board and IC development. During IC development, its socket-board environment is always considered. In a component based environment, the component pool contains the software components that are operational in different types of frameworks or environments. There is a need for a mechanism, which can rank the appropriateness of a Component in terms of its properties and its capability to operate in different environments. This paper proposes one such system model, which helps in selecting best appropriate component for an environment. As the component's usability increases, it is profiled by the proposed model. This profiling provides the indicators about its best suitability for different frameworks and operating environments.

General Terms
Component Based Software Engineering, Simulation, Component Search

Keywords
Software Component, RAD, Component Ranking, Component Attribute System (CAS).

1. INTRODUCTION
The component-based software engineering is an interesting and important field of software engineering, which helps in achieving the rapid application development. Semantically related functional modules together make one software component. These components are operational under compatible environments and increase the productivity of the system. Development environments are available using which components can be developed and maintained for future reuse. COM/DCOM, .Net, Enterprise Java Beans and CORBA are some of such development environments.

The notations and algorithms of Component storage and access are always limited to their native environment. Cross platform accessibility and functionality is very difficult to achieve. It is like using a 16 pin IC in a socket designed for 14 pin IC. The pin level detail and internal architecture details are required even for a tryout.

The development of software components is carried out in the predefined native environment and its usability is recommended in the compatible environment. The developed component resides in the distributed environment where the properties of components are used to distinguish it for a desirable environment. The component storage and accessibility mechanism do not have the intelligence or learning system, which may priorities the individual component as a best suitable candidate for the reusability request.

This paper discusses a method, which is capable of identifying the software components as the best suitable candidates against the requirement criteria. The component storage pool will become more efficient in terms of the request processing time and quality delivery of software component when there are more than one candidate components found for given requirement. The system will work on the ranking algorithms, which will update the property tables of the components. Depending upon the usability feedback from the system where the component will be used, the system will increment or decrement the numeric values of the properties of components depending upon its usability experience with that component. The proposed system categories the components for their suitability to frameworks, arrange components which can be used in different environments, components which can be used with defined cross interface notations of multiple frameworks.

The proposed system makes the component reusability system more robust, precise and efficient. Well-defined notations for the ranking of components make the system capable of learning after the component reusability. The proposed system makes it possible to provide the best possible component for the raised requirement. The source of request is used to identify the target framework and deliver the best possible component for that environment. System checks if the component is available for the same framework, as required, if yes, then it again searches for the best available components on the basis of previous usability experience (Usability ranks).

This paper explains about the experimental details and its results carried out on a simulated environment in order to achieve the precision in application development after reusing the components. The best selection criteria and ranking makes it possible for selecting the best component for raised requirements.

2. RELATED WORK

Councill and Heinemann [20] state that a software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. This shows the nature of components, and their dependency.

Crnkovic, Ivica [9] discussed about the development with components and focus on the identification of reusable entities and relations between them, starting from the system requirements. The early design process includes two essential steps: Firstly, specification of system architecture in terms of functional components and their interaction, thus giving a logical view of the systems and secondly, specification of system architecture consists of physical components [9]. This indicates a precise definition for the reusable environment.

With regard to system-wide co-ordination, components communicate with each other via interfaces. When a component offers services to the rest of the system, it adopts a provided interface, which specifies the services that other components can utilize, and how they can do so. This interface can be seen as a signature of the component - the client does not need to know about the inner workings of the component (implementation) in order to make use of it. This principle results in components being referred to as encapsulated.

Too many people are trying to make "universal" components without realizing that those components still work within some framework that allows them to be put together and communicate with each other. The problem is that other people doing the same thing have defined other "generic" frameworks that are none the less incompatible. In other words, random components don't tend to fit together or work together. Even physical components are like this. The prototypical component, the IC chip, always was designed within a family of chips that were meant to work together. They all needed the same voltage levels for zeroes and ones and tri-states, the same amperage levels, the same clock rates, etc, etc. Other families used other voltage levels [10].

Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and time-space performance. Existing sources of components - manufacturers, software houses, users' groups and algorithm collections - lack the breadth of interest or coherence of purpose to assemble more than one or two members of such families, yet software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors. The talk will examine the kinds of variability necessary in software components, ways of producing useful inventories, types of components that are ripe for such standardization, and methods of instituting pilot production [11].

Stepstone Corporation's experience amounts to an experimental study of the software components market strategy in action. The good news is that, with substantial libraries now in the field and others on the way, the chip-level software components market concept has been tried and proven sound for diverse applications in banking, insurance, factory automation, battlefield management, CAD/CAM, CASE, and many others. The component libraries have proven to be flexible enough, and easy enough to learn and use, that they have been used to build highly graphical applications in all of these domains, and sufficiently portable that it has been possible to support them on most hardware and software platforms during this era of rapid platform evolution.

The bad news is that this experiment has shown that it is exceedingly difficult, even with state-of-the-art technologies, to design and build components that are both useful and genuinely reusable, to document them so that customers can understand them, to port them to an unceasing torrent of new hardware platforms, to ensure that recent enhancements or ports haven't violated some existing interface, and to market them to a culture whose value system, encourages building everything from first principles. A particularly discouraging example of this value system is that, in spite of the time and money we've invested in libraries and environmental tools like browsers, Objective-C is still thought of as yet another programming language to be compared with Ada and C++, rather than as the tiniest part of a much larger environment of ready-to-use software components and tools [12].

All system processes are placed into separate components so that all of the data and functions inside each component are semantically related (just as with the contents of classes). Because of this principle, it is often said that components are modular and cohesive. Brad Cox [13] largely defined the modern concept of a software component. He called them Software ICs and set out to create an infrastructure and market for these components by inventing the Objective-C programming language.

Reuse has been a popular topic of debate and discussion for over 30 years in the software community. Many developers have successfully applied reuse opportunistically, e.g., by cutting and pasting code snippets from existing programs into new programs. Opportunistic reuse works fine in a limited way for individual programmers or small groups. However, it doesn't scale up across business units or enterprises to provide systematic software reuse. Systematic software reuse is a promising means to reduce development cycle time and cost, improve software quality, and leverage existing effort by constructing and applying multi-use assets like architectures, patterns, components, and frameworks.

Like many other promising techniques in the history of software, however, systematic reuse of software has not universally delivered significant improvements in quality and productivity. There have certainly been successes, e.g., sophisticated frameworks of reusable components are now available in OO languages running on many OS platforms. In general, however, these frameworks have focused on a relatively small number of domains, such as graphical user-interfaces or C++ container libraries like STL. Moreover, component reuse is often limited in practice to third-party libraries and tools, rather than being an integral part of an organization's software development processes [14].

Reusability is an important characteristic of a high-quality software component. Programmers should design and implement software components in such a way that many different programs can reuse them. Furthermore, component-based usability testing should be considered when software components directly interact with users. It takes significant efforst and awareness to write a software component that is effectively reusable. The component needs to be: fully documented, thoroughly tested, robust - with comprehensive, input-validity checking, able to pass back appropriate error messages or return codes, designed with an awareness that it will be put to unforeseen uses.

Software development is becoming less and less associated with the development, from the beginning, of a single software system and more and more an evolutionary process in which a system is incrementally developed over a series of releases. There is also an increase in the use of component-based approaches, in which the next release in the evolutionary process is defined in terms of a set of additional components that augment the existing system to meet a set of constraints. These components may be pre-existing or planned. Systems often contain an integrated mixture of preexisting components and newly built components. Much has been written about the challenges of component-based software engineering, but the focus of this work has been on the problems of integration of components, testing, defining, delineating and assuring the interfaces between components and the verification of their intended behavior and interactions [15].

Sommerville [17] explains about the component development for reuse. Components for reuse may be specially constructed by generalizing existing components. Component reusability should reflect stable domain abstractions; should hide state representation; should be as independent as possible; should publish exceptions through the component interface. There were few interesting findings about Changes for reusability - remove application-specific methods, change names to make them general, add methods to broaden coverage, make exception handling consistent, add a configuration interface for component adaptation and integrate required components to reduce dependencies .

Many techniques ranging from keyword-based to full-fledged specification-based heuristics have been proposed in the literature to provide effective retrieval of qualified components during the discovery process. The keyword-based approach is simple and flexible as users simply specify the query as a set of keywords representing the component requirements in which they are interested. This approach while simple is also prone to low accuracy resulting in either too many or too few hits, or in some cases even completely unrelated hits. The faceted approach classifies components based on predefined taxonomies. While this approach provides a better description of components than a pure keyword-based approach, users must be familiar with the classification scheme to effectively retrieve a needed component. Moreover, it is often hard to manage classification schemes when domain knowledge evolves and as a result the component falls into two or more categories. Signature matching approaches decide the match between two given components, the query and library components, based on the signatures of the methods in these two . While signature matching uses intrinsic built-in information about the component, that is its type information, it often still returns irrelevant hits. For example, consider the methods strcpy and strcat in the standard C library. These methods have the same signature but encode different behaviors. The specification matching approach, introduced to overcome the problem of signature matching, uses the method’s pre- and post-conditions that capture the functionality of the method. While specification matching provides more accurate hits, it is too time-consuming to be practical as its implementation, often based on theorem proving techniques, is expensive. Another drawback of the specification approach is the practical lack of pre- and post-conditions in component code [18].

3. PROBLEM STATEMENT

The performance of component based software development approach is dependent upon how well the component is delivered from the component repository when required. It has been observed that there is no standard way of placing the component in some repository after development. The component accessing mechanism is also not well defined. There are many methods available that can get us the list of components available in component libraries. But problem with these existing techniques is that they can only give us abstract view of the component features. For details we have to put in extra efforts and go through the detailed associated document. There is a need for the component storage and access mechanism, which can assist the software development process for good quality software product and at a faster rate.

There is a need for a system which can add attribute to the components, information about where they can be well integrated and perform well, information about the behavior details of component, information about the pre and post conditions and information about a component if that can be used in cross platform environment.

Even the components that give the same behavior and same functionality also require the ordering in terms of best suitable component for the requirement. This parameter comes from the system that uses the components and then after using, also provides the rating for the used components. If some component is used in some project and performs well, then its associated attributes need to be updated so that it may become the best component for a given requirement. Likewise, using the component storage and access system during the project development, it is also possible that a component may not fulfill the functionality that it claims. In that case, its respective attribute values are decremented and system rates-down that component. In future, use of that component will be less recommended by the system. So in this way components can be ranked according to their ability to fulfill certain requirements.

These are few of requirements which needs to be there in order to use and manage the component based software development

4. EXPERIMENTAL METHODOLOGY

The experiment conducted to study the precision in reusability of reusable components uses the attributes assigned to the components by the proposed Component Attribute System [CAS]. The experiment uses thirty-five components and those were used in the different requirements. The system also uses a system to store and access the components in the component repository [19]. During the experiment, the judgment was on the bases of user experience. For the new project requirement, two approaches were followed: 1) Using the CAS based system for component accessing and reusability and other was 2) using traditional approach of using the components.

The proposed system uses the component's property and stores it using the XML based attribute assignment with each component. The keywords are required to specify the details about the platform for which the component is being developed, the details about the core functionality of the module, the pre-post condition, component handles in terms of the input parameters and the output value details. These exposed keywords with each component play important role while evaluating the exiting component by specifying a set of keywords. The search keywords give the details about the platform for which the component is required, the details about the core functionality of the desired module, the pre-post condition, component handles in terms of the input parameters and the output value type.

There is system that updates the component ranks (as shown in Figure 3) to Component Attribute System.

There may be two methods of doing this. One is that the system that is going to reuse the existing component may be requested to provide the feedback. Feedback may be provided by the system automatically or the programmers working on the new system may provide the feedback. Another method may be that a counter is associated with each component. As and when that component is chosen for reuse by some new system, the counter is incremented automatically. This will help the future searches in identifying the best suitable components for their type of applications.

The other functionality of the proposed system is that the expert analysis is possible on the stored repository of the components. The experts/developers take the component and then checks for the components claims. If the component is not found good against its claims, the expert may rate it poor and can send their opinion to the system. The authorized rating are considered and

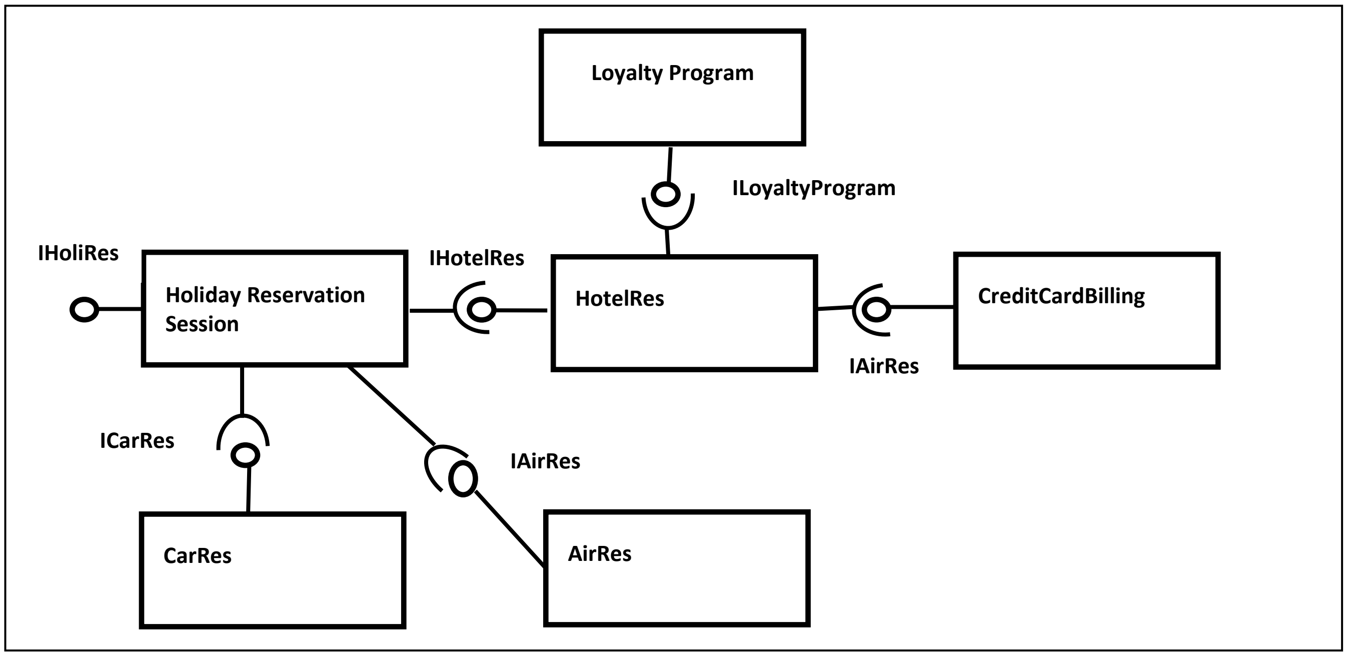

Fig 1: Reusable components arranged after CAS search

The search process takes the keywords as input and matches them with the attributes of the available components. The keyword matching also takes care of the maturity of components for a set of requirement. The more often a component is used, and has more recommendation by the CAS system for fulfilling the requirement, the more mature the component becomes. The CAS recommendation system takes care about the feedback and depending upon the rating it makes component as a strong candidate for the reusability. When we assign some keywords for searching a component, it searches for requirement and then sorts the available components on the basis of reusability rank. The component with best rank can be used after identifying its interfaces. The system that reuses the component will be added with the functionality of feedback that responds for reusability status after some component has been reused. It also interacts with the user to collect the experience.

the component ranks may increase or decrease depending upon the type of expert rating.

Reusable Components of one such system arranged after CAS search are shown in Fig 1. It can be seen in the figure that each component may have two types of interfaces, one is 'provide' interface through which it provide services to other components of the system and the second is 'Gets' interface through which it takes services from other components in the system. For example 'HotelRes' Component has two 'Gets' interfaces through which it take service from 'LoyaltyProgram' and 'CreditCardBilling' components respectively and one 'Provide' interface through which it provide services to 'Holiday Reservation Session' component.

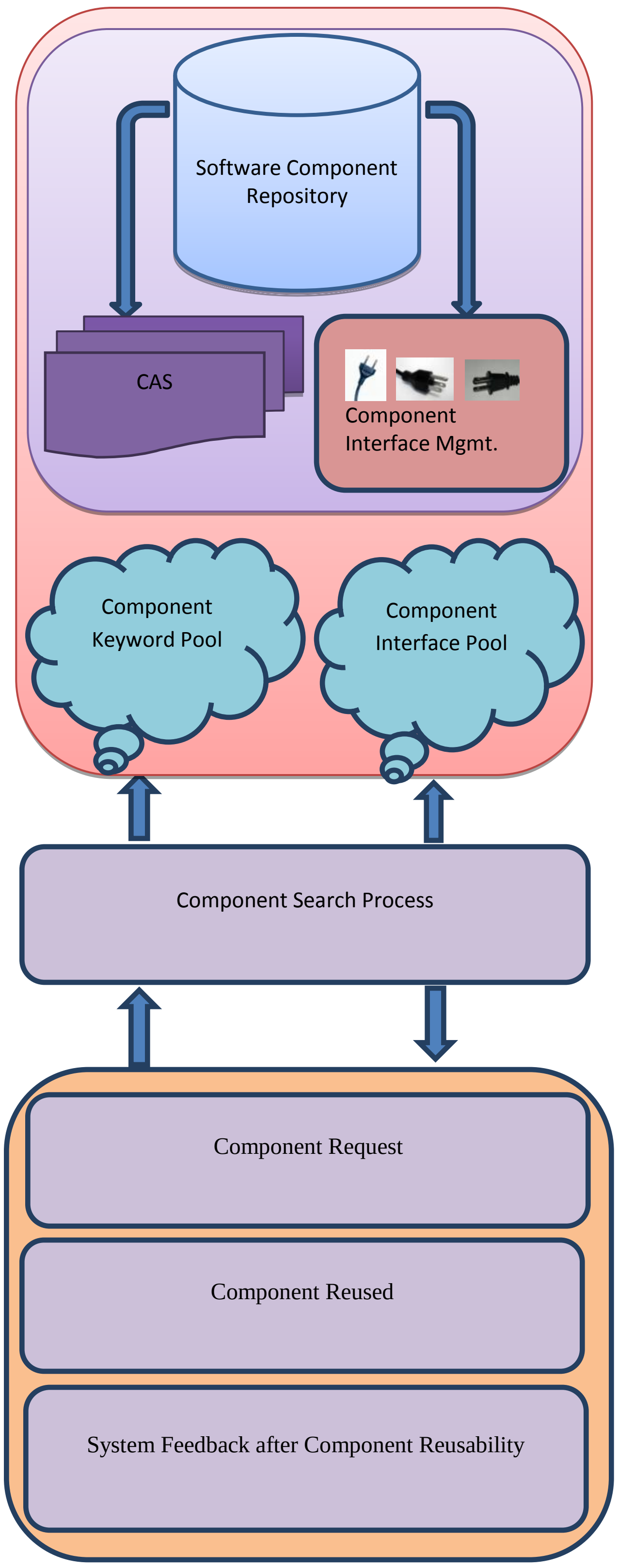

Fig 2: Components lookup and delivery using CAS

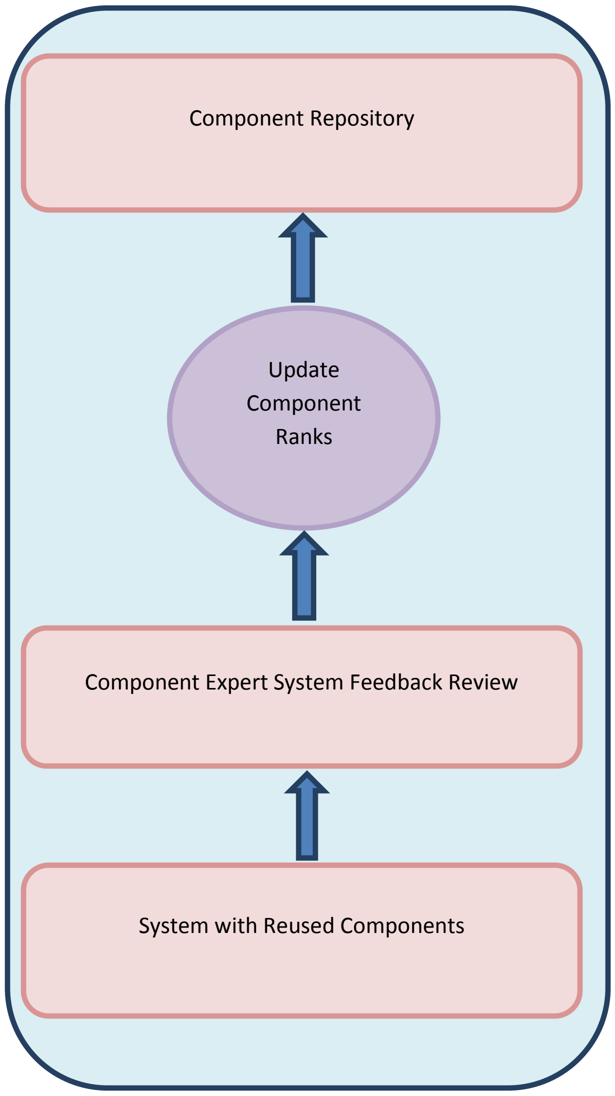
Component Repository
Update Component Ranks
Component Expert System Feedback Review
System with Reused Components

Fig 3: Components rank update after system feedback or expert review

5. RESULTS AND ANALYSIS

The results obtained after experimenting with the Software components by attaching the attributes to Components and defining the interface structure are very useful. It was observed that the proposed system can be used with a system where we have small component repository so as of 1000 components to a system where the number of components exceed to very large . The experiments shows the two different types of observations as 1) improvements in user experience which make the proposed system very useful by reducing the time consumed to search a reusable component.

A comparative analysis of results is shown in table 1 and 2. Table 1 compares the time taken by a search process that uses CAS and a process that does not use the CAS. It was observed that time taken by CAS based search is much less. These results are also depicted in Fig 4.

Table 1. Comparison of time taken by search process using CAS and without using CAS

Simulation Experiment	Reusable component type	User experience with CAS (Time in min)	User experience without CAS
1	Airlines reservation	11	30
2	Credit card billing	17	123
3	Bar code reader	14	78
4	Report viewer	6	67
5	Bar code generator	12	34
6	Phone book	16	88
7	Scheduler	12	56
8	Book search	7	27
9	Logger (To Excel)	5	78
10	Word file reader	13	102
11	Alarm	16	133

Table 2. Comparison of the level of precision achieved using CAS and without using CAS

Simulation Experiment	Reusable component Type	Level of appropriateness with CAS	Level of appropriateness without CAS
1	Airlines reservation	48	10
2	Credit card billing	78	16
3	Bar code reader	62	13
4	Report viewer	66	8
5	Bar code generator	89	13
6	Phone book	43	17
7	Scheduler	58	19
8	Book search	49	7
9	Logger (To Excel)	23	9
10	Word file reader	82	11
11	Alarm	53	13

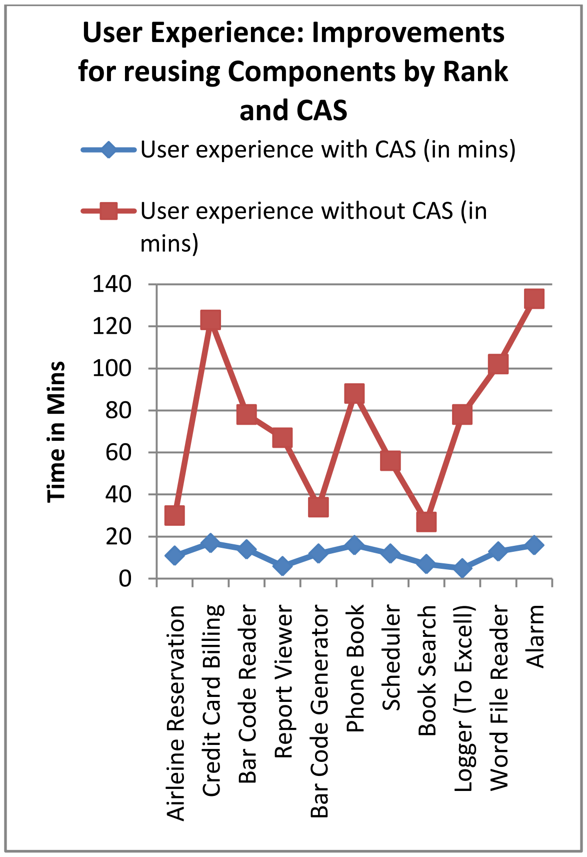

Fig 4: Improvement for reusing Components by Rank and CAS

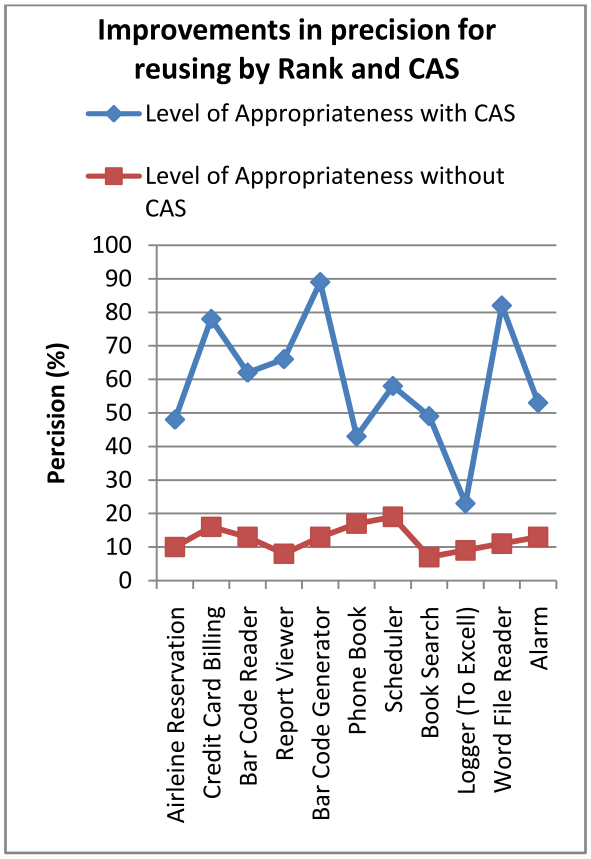

Fig 5: Improvement in precision for reusing Components by Rank and CAS

The reduction in time makes the system more productive and useable. But time saving for building the reusable system is not the only objective. The experiment with the proposed system shows the level of precision achieved by using the proposed approach. These observations are compared in table 2. It lists the "Level of appropriateness" using CAS and without using CAS. The level of appropriateness can be measured in terms of ratio of the total features found in the searched component to the number of features a search process was looking for in the component.

The percentage is shown as the level of appropriateness. Results show that level of appropriateness is greater while using the CAS based search. Results are depicted with the help of Fig 5 also. The system ensures that the reusable component is made available to user in less time with more appropriateness towards the requirements. Therefore, the propose system is capable to improve the overall performance of component based system by adding the suitable attributes and interfaces.

6. CONCLUSION

Overall it is concluded that although there are many techniques available that can be used to search for the reusable components from component libraries, but they lack in many aspects like time taken in searching the component and finding the appropriate component to be reused. The proposed model named as CAS (Component Access System) ensures that the reusable component is made available to the users in less time with more appropriateness towards the requirements. Therefore the proposed system is capable of improving the overall performance of Component Based system by adding the suitable attributes and interfaces.

7. REFERENCES

[1] Boehm, B. W. 1984. Software engineering economics. IEEE Transactions on Software Engineering. 10(1):4-21.

[2] Burgess, C. J. and M. Lefley. 2001. Can genetic programming improve software effort estimation? A comparative evaluation. Information & Software Technology .43(14): 863-873.

[3] Wong, W. E., Horgan, J. R., London S., and Bellcore, H.E. 1997. A study of effective regression testing in practice. In ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering. IEEE Computer Society. Page 264.

[4] Srivastava, A. and Thiagarajan, J. 2002. Effectively prioritizing tests in development environment. Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis (ISSTA-02). Volume 27, 4 of Software Engineer Notes. 97–106. ACM Press. New York.

[5] Antoniol, G., Penta, M.D. and Harman, M. 2005. Search Based techniques Applied to Optimization of Project Planning for a Massive Maintenance Project. In 21st IEEE International Conference on Software Maintenance, Los Alamitos, California, USA. 240–249. IEEE Computer Society Press.

[6] Clark, J., Dolado, J.J., Harman, M., Hierons, R.M., Jones, B., Lumkin, M., Mitchell, B. Mancoridis, S. Rees, K., Roper, M. and Shepperd, M. 2003. Reformulating Software Engineering as a Search Problem. IEE Proceedings — Software, 150(3):161–175.

[7] Kirsopp, C., Shepperd, M and Hart, J. Search Heuristics, Case-Based Reasoning and Software Project Effort Prediction. 2002. In GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. 1367–1374, San Francisco, CA 94104, USA. Morgan Kaufmann Publishers.

[8] Sitaraman M. and Weide B. W. 1994. Special Feature: Component-Based Software Using RESOLVE. ACM SIGSOFT Software Engineering Notes 19, No. 4. 21-67.

[9] Crnkovic, I. Component-based Software Engineering – New Challenges in Software Development, Mälardalen University, Department of Computer Engineering, Västerås, Sweden.

[10] Wallace, B. 2000. There is no such thing as a Component. PolyGlot publication, San Francisco, Ca, United States.

[11] Naur, P. and Randell, B. 1969. Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany. Scientific Affairs Division, NATO, Brussels. 138-155.

[12] Cox, B.J. 1990. Planning the Software Industrial Revolution. Software Technologies of the 1990's, IEEE Software magazine.

[13] Cox's feasibility demonstration of a usage-based mechanism for incentivizing component producers, http://virtualschool.edu/mybank.

[14] Schmidt, D. C. 1999. Why Software Reuse has failed and How to Make It Work for You. C++ Report magazine. Department of Electrical and Computer Engineering, University of California, Irvine.

[15] Baker, P., Harman, M., Steinh¨ofel, K. and Skaliotis, A. 2006. Search Based Approaches to Component Selection and Prioritization for the Next Release Problem. Motorola Labs., Viables Estate, Basingstoke, Software Maintenance. ICSM '06. 22nd IEEE International Conference on 24-27 Sept.176-185. Philadelphia, PA, USA.

[16] Rothermel, G., Elbaum, S., Malishevsky, A.G., Kallakuri, P. and Qiu, X.2004. On test suite composition and cost-effective regression testing. ACM Trans. Software Engineering Methodologies. 277–331.

[17] Sommerville, I. Component-based software engineering, Software Engineering, 7th edition. Chapter 19.

[18] Naiyana, T. and Kajal, C. 2005. Finding a needle in the haystack: A technique for ranking matches between components. Lecture notes in computer science ISSN 0302-9743, Congress CBSE 2005: component-based software engineering, International symposium No 8, St. Louis, MO. Vol. 3489. 171-186. ISBN 3-540-25877-9.

[19] Suri P. K. and Singh G., 2010. Framework to represent the software design elements in markup text – Design Markup Language (DGML). IJCSNS International Journal of Computer Science and Network Security, South Korea, VOL.10 No.1, January. 164-170.

[20] Heinemann G.T. and Councill W.T. Component Based Software Engineering- Putting the Pieces Together. Addison Wesley Publisher.