Research Paper

Volume : 3 | Issue : 2 | February 2014 • ISSN No 2277 - 8179

Software Reliability Analysis- A New Approach

Computer Science

KEYWORDS : CORBA, Software Reliability, Component-based Software Engineering

Taraq Hussain Sheakh Research Scholar JJTU Rajasthan

ABSTRACT

New reliability models are used in order to estimates the reliability of the software depends upon the component structure of code. The analyses of these models emphasize the dependence of component failures, component based testing and component based implementation. These models can provide quality based assurance of the components reuse in the system and also the component of the system are provide feasibility of software reliability of this type are important for two main reasons. Traditionally, the complexity of a computation is measured in terms of the number of elemental computations used. In contrast, a statistical complexity measure is proposed to describe complexity in terms of statistical software testability. A highly complex program requires intense testing in order to justify a claim that the program achieves a given level of reliability. A low complexity program requires less testing to achieve the same claim. In this Research paper I tend to emphasize the analysis of the software reliability on the basis of the components of the system.

2. INTRODUCTION

The concept of reuse is very important phenomenon in core engineering branches. Today complex, high quality software systems are built efficiently using component based approach in a short period of time. But a number of questions arise about the feasibility of the component approach. But the concept of CBD successfully answers the arising questions. The importance of component Based development lies in its efficiency. It takes only a few minutes to assemble the original system because the components are intended to be cohesive with affluence. Although software is substantially more complex, it follows that component-based systems are easier to assemble and therefore less costly to build than systems constructed from isolated parts. In addition, CBSE encourages the use of expectable architectural patterns and standard software infrastructure, thereby leading to a higher-quality result.

1.1Objectives of Component Based Software Engineering:
The main objectives of component based software engineering are given below.

a). Reduction of cost and time for edifice large and complicated systems: main objective of Component based approach is to build intricate software systems using off the shelf component so that the time to build the software moderate drastically. The cost effectiveness of the current method can be analysed using function point or other methods.
b). Improving the quality of the software: The quality of the software can be enhanced by improving the quality of the component. Though the concept is not true in general. Sometimes quality of the assembled systems may not be directly related to quality of the component in sense that improving the quality of the component does not necessarily imply the improvement of the systems.
c). Detection of defect within the systems: Component approach helps the system to find its defect readily by testing the components. But the source of defects is difficult to find in case of component development approach.

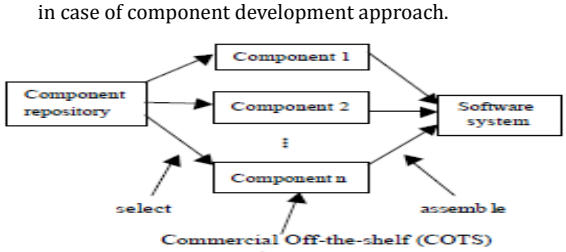

Figure 1. Component-based software development

2. Component-based software engineering
Software complexity has dramatically increased since the mid-1980s (Gao et al. 2003,5). Nowadays, new systems are seldom developed from scratch, so software developers need cost-effective methods to construct complicated software systems. The software business is moving increasingly towards component-based software development (Brown & Wallnau, 1996). *Component-based software engineering*(CBSE) shifts the emphasis from programming software to composing software systems (Clements, 1995).

In CBSE, complex systems are constructed by assembling them from software components according to a software architecture. In CBSE it is impossible to build a coherent system of interworking components without a comprehensive architectural model (Klingler, 2000). The s*oftware architecture* of a system can consist of several structures of the system, and is composed of software elements, the externally visible properties of those elements, and the relationships between them (Bass et al. 2003, 21). Some of the most common architectural structures of a system are decomposition, layered, process, deployment, and client server.

The advantages of a component-based approach are the possibility to master development and deployment complexity, modularity, decreased time to market, the quality and reusability of software and its components, the composed services of components, and the scalability and adaptability of software systems (Herzum & Sims, 2000, 21-23). Furthermore, software suppliers can specialize in their strategic competitive edge and buy other properties as ready-made COTS (commercial-off-theshelf) components. The greatest challenges in component-based software development are that suitable ready-made components cannot be easily found, and if they are found, they are not necessarily of good quality (Vitharana et al. 2003). In addition, components have different dependencies between them, source code is seldom available, and the target context of the components is not known. Maxville et al. have described a process to help an integrator to select the right components from a huge amount of third party components (Maxville et al. 2003). The method has the following phases: problem definition and requirements for the component (ideal component specification), short-list creation of candidate components, test case generation based on the ideal component specification, test adaptation, test execution, General concepts in component-based software development evaluation of results, candidates' ranking based on the results and other suitability and context information, and reporting on the results.

It is a manual process but their aim is to automate it as much as possible. The other method for component selection is called *a trusted third party*, which requires different roles to support component-based software development and each of these roles are responsible for the quality of the component (Stafford & Wallnau, 2001). The basic roles are component technology specified, component technology implementer, reasoning framework developer, component implementer, and system developer.

A software component can be deployed independently and is subject to composition by third parties." Thus, the components interoperate with each other through interfaces. An *interface* defines the access points of the component and allows other components to use services provided by the component (Szyperski, 2002, 42-43). Components can have different types of interfaces (Sametinger, 1997, 71-76): a programming interface, a data interface, a user interface, and/or an interface to the infrastructure. To use the interfaces, contracts are needed. A *contract* states what the client needs to do to use the interface and what the provider has to implement to meet the services promised by the interface (Szyperski, 2002). Contracts can be divided into four levels: basic or syntactic, behavioural, synchronization, and quality-of-service (Beugnard et al. 1999). *Basic contracts* specify the operations that a component performs, the input and output parameters each component requires, and the possible exceptions that might be raised during operations.

A *behavioral contract* specifies the behaviour of operations more precisely than a basic contract. It sets out preconditions, post conditions, and class invariants. A *precondition* has to be met before the execution of an operation. A *post condition* has to be met just after the completion of an operation. A *class invariant* will always keep its truth value throughout a specific sequence of operations: it constrains objects of a class and the state stored in an object. A *synchronization contract* specifies the dependencies between the services provided by a component. A *quality-of-service contract* specifies the quality features the service will respect, such as maximum response delay, average response, and the quality of the result.

2.1 Testing component-based systems

A component-based system is not monolithic: it contains components of different granularities (e.g. lowest level components, business components, and component based systems), which are integrated with other components and into legacy systems with interfaces. In such situations, testing and documentation are even more important than in conventional software projects with monolithic applications. When moving from legacy systems to component-based systems, interfaces and interface testing are needed. The components do not have to know each other's implementation, only the content of the interfaces, i.e. syntax, semantics, and instructions for using the interface.

The components can be implemented by the integrator or they can be acquired ready-made and then integrated into the one's own systems. If components are re-used in a new context, regression testing .However, it is often forgotten that testing activities depend on who is performing testing, and thus on the availability and quality of the specifications and documentations. A tester may be a component developer, a Component integrator or a customer (end user). The component developer usually has all the needed testing materials, so for the developer the component is a white box, and code-based and specifications-based testing techniques can be used. The integrator and customer seldom have a source code available. Thus, the component is a black-box with interfaces, and test cases must be developed based on the other available documents, such as design and interface specifications, user manuals, and contracts. Integration has two parallel parts: on the one hand, user interface components are integrated and tested, and on the other hand business logic and resource components are integrated and tested. Furthermore, the components' external interfaces have to be tested with operation calls, i.e. how the component behaves if it is called outside the component. Dependencies between components can be controlled with the help of a dependency graph, which shows the dependencies between the components of the same granularity level, and assures that the whole functionality of the component has been covered by test cases. Without the dependency graph, some critical paths may remain non-executed. More details about dependency graphs and an example of dependency graph.

1.2 CORBA Component Model (CCM):

CORBA Component Model (CCM) is an addition to the family of CORBA definitions. It was introduced with CORBA 3 and it describes a standard application framework for CORBA components. Though not dependent on "language independent Enterprise Java Beans (EJB)", it is a more general form of EJB, providing four component types instead of the two that EJB defines. It provides an abstraction of entities that can provide and accept services through well-defined named interfaces called ports. The CCM has a component container, where software components can be deployed. The container offers a set of services that the components can use. These services include (but are not limited to) notification, authentication, persistence and transaction processing. These are the most-used services any distributed system requires, and, by moving the implementation of these services from the software components to the component container, the complexity of the components is dramatically used.

3. Comparison among Current omponent Technologies

Comparison among current component echnologies can be found in [Brow98], [Pour99a] and [Szyp98]. Herewe simply summarize these different features in Table 1

	CORBA	EJB	COM/DCOM
Development environment	Underdeveloped	Emerging	Supported by a wide range of strong development environments
Binary interfacing standard	Not binary standards	Based on COM; Java specific	A binary standard for component interaction is the heart of COM
Compatibility & portability	Particularly strong in standardizing language bindings; but not so portable	Portable by Java language specification; but not very compatible.	Not having any concept of source-level standard of standard language binding.
Modification & maintenance	CORBA IDL for defining component interfaces, need extra modification & maintenance	Not involving IDL files, defining interfaces between component and container. Easier modification & maintenance.	Microsoft IDL for defining component interfaces, need extra modification & maintenance
Services provided	A full set of standardized services; lack of implementations	Neither standardized nor implemented	Recently supplemented by a number of key services
Platform dependency	Platform independent	Platform independent	Platform dependent
Language dependency	Language independent	Language dependent	Language independent
Implementation	Strongest for traditional enterprise computing	Strongest on general Web clients.	Strongest on the traditional desktop applications

Table 1. Comparison of current component technologies

4. Quality Characteristics of Components

As much work is yet to be done for component-based software development, QA technologies for component based software development has to address the two inseparable parts: 1) How to certify quality of a component? 2) How to certify quality of software systems based on components? To answer the questions, models should be promoted to define the overall quality control of components and systems; metrics should be found to measure the size, complexity, reusability and reliability of components and systems; and tools should be decided to test the existing components and systems. To evaluate a component, we must determine how to certify the quality of the component. The quality characteristics of components are the foundation to guarantee the quality of the components, and thus the foundation to guarantee the quality of the whole component-based software systems. Here we suggest a list of recommended characteristics for the quality of components: 1) Functionality; 2) Interface; 3) Usability; 4) Testability; 5) Maintainability; 6) Reliability. Software metrics can be proposed to measure software complexity and assure its quality [16] [17]. Such metrics often used to classify components include [6]:

1) **Size.** This affects both reuse cost and quality. If it is too small, the benefits will not exceed the cost of managing it. If it is too large, it is hard to have high quality.
2) **Complexity.** This also affects reuse cost and quality. A too-trivial component is not profitable to reuse while a too-complex component is hard to inherit high quality.
3) **Reuse frequency.** The number of incidences where a component is used is a solid indicator of its usefulness.
4) **Reliability.** The probability of failure-free operations of a component under certain operational scenarios.

5. CONCLUSIONS

The aim of this paper is to analyse the various techniques which is based upon the component models. The new approach using the component reliability methods is very effective regarding the reuse. It is also mentioned that the reliability, quality of the predictability of the system reliability is also be enhanced using the components based reliability model. This further emphasise that in the today complex and large line of coding based system the component reliability model is best for the throughput enhancement and the same the reliability of the system. The cost as well as the time of using the component reliability models are also meagre since the use of components for black box testing or testing is implemented in easiest manner.

REFERENCE

[1] Hughes G, May JHR, Lunn AD "Reliability estimation from appropriate testing of plant protection software", IEE Software Engineering Journal, v10 n6, November 1995. | [2] Goseva-Popstojanova K, & Trivedi K.S. "Architecture-based approach to reliability assessment of software systems"Performance Evaluation, v45 n2-3, July 2001. | [3] Kanoun K, Laprie JC, Thevenod-Fosse P "Software reliability: state-of-the-art and perspectives" LAAS Report.95205, May 1995. http://www.laas.fr/laasve/ | [4] Kuball S., May J.H.R. & Hughes G. "Structural software reliability estimation," Lecture Notes in Computer Science v1698 'Computer Safety, Reliability and Security' Felici, Kanoun, Pasquini (Eds) pp336-349 ISBN 3540664882 Springer 1999. | [5] Kuball S., May J.H.R. & Hughes G. "Building a System Failure Rate Estimator by Identifying Component FailureRates" Procs. Of the 10th Int. Symposium on Software Reliability Engineering (ISSRE'99, Boca Raton, Florida, Nov 1-4, 1999) pp32-41 IEEE Computer Society 1999. | [6] Kuball S., May J.H.R. & Hughes G. "Software Reliability Assessment for Branching Structures: A Hierarchical.Approach," 2nd Int. Conference on Mathematical Methods in Reliability, vol 2, Bordeaux, July 4-7 2000 [7] Lyu MR "Handbook of Software Reliability Engineering," McGraw-Hill 1996. | [8] May JHR, Lunn AD "A Model of Code Sharing for Estimating Software Failure on Demand Probabilities" IEEE Trans. on Software Engineering SE-21 (9) 1995. | [9] May JHR and Lunn AD "New Statistics for Demand-Based Software Testing" Information Processing Letters 53, 1995. | [10]May JHR, Kuball S, Hughes G "Test Statistics for System Design Failure" International Journal of Reliability, Qualityand Safety Engineering, v6 n3 1999 pp. 249-264. | [11] Miller WM , Morell LJ, Noonan RE, Park SK, Nicol DM, Murrill BW and Voas JM "Estimating the probability of failure when testing reveals no failures" IEEE Trans. On Software Engineering v18 n1 1992 | [12]Musa JD "Operational profiles in software reliability engineering," IEEE Software 10(2) 1993 | [13]Thevenod-Fosse P, "On the efficiency of statistical testing with respect to software structural test criteria" IFIP Working Conference on Approving Software Products, Garmisch (Germany), 17-19 Sept 1990, pp.29-42. Also LAAS Report 90156, May 1990. | [14]Zhu H, Hall PAV, May JHR "Software Unit Test Coverage and Adequacy," ACM Computing Surveys 1997. | [15] Roger Pressman, Software Engineering: A Practitioner's Approach, Fifth Edition. | [16] Robert Orfail, Dan Harkey Client / Server Programming with CORBA, John Wiley & Sons | [17] P.J. Deitel, H.M.Deitel, JAVA for Programmers *4+ Wang, Schimdt, O'Ryan Overview of the CORBA Model, Wiley & Sons 2000. | [18] Crnkovic, Larsson, Michel Chaudron Component-based Development Process and Component Lifecycle, ABB Corporate Research , Vasteras, Sweeden. | [19] Jifeng, Xiaoshan Li, Zhiming Liu Component Based Software Engineering – The Need to Link Method and their Theories, UNU IIST Report No 330. . | [20] Martin Wirsing, A Component Based Approach to Adaptive User-centric Pervasive , 13th International Symposium on Component Based Software Engineering, 2010 | [21] Stephen Thesing, The Joy of Qualifying Siftware in the Avionics Area, 13th International Symposium on Component Based Software Engineering, 2010. |