Testing-Effort Dependent Software Reliability Model for Distributed Systems

Omar Shatnawi, Department of Computer Science, Al al-Bayt University, Mafraq, Jordan

ABSTRACT

Distributed systems are being developed in the context of the client-server architecture. Client-server architectures dominate the landscape of computer-based systems. Client-server systems are developed using the classical software engineering activities. Developing distributed systems is an activity that consumes time and resources. Even if the degree of automation of software development activities increased, resources are an important limitation. Reusability is widely believed to be a key direction to improving software development productivity and quality. Software metrics are needed to identify the place where resources are needed; they are an extremely important source of information for decision making. In this paper, an attempt has been made to describe the relationship between the calendar time, the fault removal process and the testing-effort consumption in a distributed development environment. Software fault removal phenomena and testing-effort expenditures are described by a non-homogenous Poisson process (NHPP) and testing-effort curves respectively. Actual software reliability data cited in literature have been used to demonstrate the proposed model. The results are fairly encouraging.

Keywords: Distributed Development Environment, Fault Severity, Software Component, Software Engineering, Software Reliability, Software Testing-Effort

INTRODUCTION

The software development environment is changing from a host-concentrated one to a distributed one due to cost and quality aspects and rapid growth in network computing technologies. Distributed systems are being developed in the context of the client-server architecture. Client-server architectures dominate the landscape of computer-based systems. Everything from automatic teller networks to the Internet exists because software residing on one computer—the client—requests services and/or data from another computer—the server. Client-server software engineering blends conventional principles, concepts, and methods with element of object-oriented and computer-based software engineering to create client-server systems. Client-server systems are developed using the classical software engineering activities.

Distributed systems are growing rapidly in response to the improvement of computer hardware and software and this is matched by
the evolution of the technologies involved (Zhao et al., 2010). Developing large-scale distributed systems (LSDS) is complex, time-consuming, and expensive. LSDS developments are now common in air traffic control, telecommunications, defense and space. In these systems a release will often be developed over 2-4 years and cost in excess of 200-300 person years of development effort (Kapur et al., 2004b). Due to their complexity, LSDS are hardly ever “perfect” (Lavinia et al., 2011). Features define the content of a release. The features are realized by mapping the full set of system feature requirements across the various components. Normally this involves building on a large existing software base made up of the existing components that are then modified and extended to engineer the new release. Requirement changes are common in these developments because of the long lead times involved. Change is inevitable since user requirements, component interfaces, and developers’ understanding of their application domain all change. This adds to the complexity of planning and controlling the in-progress development both at the individual component and the release. A further complexity factor is the integration and validation of the release once the individual components are delivered. These large-scale systems must achieve high reliability because of their nature. In the final release validation software defects arise not only from the new software but also due to latent defects in the existing code. Substantial regression tests must be run to checkout the existing software base that may amount to millions of lines of code (Kapur et al., 2004b).

Successful operation of any computer system depends largely on its software components. Thus, it is very important to ensure the quality of the underlying software in the sense that it performs its functions that it is designed and built for. To express the quality of the software to the end users, some objective attributes such as reliability and availability should be measured. Software reliability is the most dynamic quality attribute (metric), which can measure and predict the operational quality of the software. Software reliability model (SRM) is the tool, which can be used to evaluate the software quantitatively, develop test cases, schedule status and monitor the change in reliability performance. In particular, SRMs that describe software failure occurrence or fault removal phenomenon in the system testing phase are called software reliability growth models (SRGMs). Among others, non-homogeneous Poisson process (NHPP) models can be easily applied in the actual software development.

Some SRGMs are concerned with the cumulative number of faults detected by software testing and the testing period (Lyu, 1996; Xie, 1991; Musa, 1999; Kapur et al., 1999; Pham, 2000; Shatnawi, 2009a). These models assume testing-effort to be constant throughout the testing period. Other SRGMs have incorporated the expenditures due to testing-effort (Yamada et al., 1985; Kuo et al., 2001; Kapur & Bardhan, 2002; Kapur et al., 2006; Huang et al., 2007; Kapur et al., 2008). They assumed that the fault detection rate is proportional to current fault-content and software testing-effort expenditures. Testing-effort expenditures are the resources spent on software testing. It has been observed that the relationship between the testing time and the corresponding number of faults removed is either Exponential or S-shaped. An interesting inference can be made regarding the analysis namely all the models are robust and can be used for any testing environment and can be termed as Black-Box models, which are used without having any information about the nature of the software being tested. However, if one has to develop what is called White-Box model, one needs to know about the software technology, which has been used to develop the software. Thus, it is imperative to clearly understand the software development environment and accordingly there is need to develop a model, which can explicitly explain the software technology that has been used to develop the software and now being tested. Such a modelling approach was earlier adopted by (Pham, 2000; Shatnawi, 2009b; Kapur et al., 2004a; Ohba, 1984; Yamada et al., 1985). Such approach is very much suited for object-oriented programming and distributed development environments (Kapur et al., 2004a; Shatnawi, 2009b).
[www.igi-global.com/article/g2g-meta-grid-framework-convergence/45743?camid=4v1a](www.igi-global.com/article/g2g-meta-grid-framework-convergence/45743?camid=4v1a)

Effective Resource Allocation and Job Scheduling Mechanisms for Load Sharing in a Computational Grid
[www.igi-global.com/chapter/effective-resource-allocation-job-scheduling/20506?camid=4v1a](www.igi-global.com/chapter/effective-resource-allocation-job-scheduling/20506?camid=4v1a)