# On the Need for Careful Definition of and Improved Capabilities in Quality-of-Service Unit Testing

James H. Hill
Vanderbilt University
Nashville, TN, USA
j.hill@vanderbilt.edu

## Abstract

*Unit testing traditionally is a process for increasing confidence levels in functional attributes of large-scale distributed systems throughout the software lifecycle. As large-scale distributed systems grow larger and more complex, increasing confidence levels in their quality-of-service (QoS) attributes, such as performance, reliability, and security, throughout the software lifecycle is becoming increasingly important. Little work, however, has investigated the challenges associated with unit testing QoS attributes of large-scale distributed systems throughout the software lifecycle.*

*This paper provides two contributions to testing QoS attributes of large-scale distributed systems. First, this paper defines the meaning unit testing QoS attributes. Secondly, it discusses challenges associated with unit testing QoS attributes of large-scale distributed. By addressing the challenges presented in this paper, distributed system developers will be able to improve QoS assurance of large-scale distributed systems throughout the software lifecycle.*

## 1 Introduction

**Challenges of testing large-scale distributed systems.** Unit testing [6, 8] is the process of evaluating functional attributes, such as build and behavior correctness, of individual units (or pieces) of a software system, such as methods of a class or a single component, in isolation. Unit testing helps increase confidence levels in quality assurance that the software system functions properly. For example, when a unit test passes, system developers are more confident that the software unit under test will function as expected when integrated into the software system. When a unit test fails, system developers are more aware that portions of the software system that utilize that unit of software may not function properly. Moreover, when unit testing is integrated with continuous integration environments, such as CruiseControl (`cruisecontrol.sourceforge.net`), this helps increase confidence levels in quality assurance continuously throughout the software lifecycle.

As large-scale distributed systems grow larger and more complex (*e.g.*, ultra-large-scale systems [7]) it is becoming particularly important to unit test functional attributes of such systems continuously throughout the software lifecycle. In addition to satisfying functional attributes, large-scale distributed systems must also satisfy quality-of-service (QoS) attributes (*e.g.*, availability, performance, reliability, and security). Evaluation of QoS attributes for large-scale distributed systems throughout the software lifecycle, however, is hard due in part to the *serialized-phasing development* problem [11], where infrastructure- and application-level software system entities are developed sequentially and QoS evaluation cannot begin until late in the software lifecycle, *i.e.*, at system integration time.

System execution modeling (SEM) tools [12] help address the serialized-phasing development problem by enabling distributed system developers to conduct system integration test on the target architecture during early stages of the software lifecycle, *i.e.*, before system integration time. Although SEM tools facilitate early integration testing, SEM tools do not provide techniques for evaluating QoS attributes of large-scale distributed systems continuously throughout the software lifecycle. Instead, distributed system developers must rely on *ad hoc* techniques, such as handcrafted cron jobs or execution and analysis scripts, that must be (re)invented when applied across different application domains. Distributed system developers therefore need new methodologies that will enable them to evaluate and reason about QoS attributes in a manner similar functional attributes of large-scale distributed systems.

**Solution approach → Unit testing QoS attributes.** Unit testing QoS attributes of large-scale distributed systems is similar to unit testing functional attributes in that individual units of a software system are evaluated to increase confidence levels. When unit testing QoS attributes, however, QoS attributes (such as performance, reliability,

and security) of the software unit are evaluated in conjunction with functional attributes (such as build and behavior correctness). This enables distributed system developers to increase confidence levels that the software system will meet QoS requirements continuously throughout the software lifecycle.

*Understanding Non-functional Intensions via Testing and Experimentation (UNITE) [5]* presented a methodology for enabling distributed system developers to unit testing QoS (*i.e.*, non-functional) attributes of large-scale distributed systems. Although UNITE enables unit testing QoS attributes, the actual definition of unit testing QoS attributes is still undefined. This paper therefore presents a definition for unit testing QoS attributes and its associated challenges. Moreover, by addressing these challenges, distributed system developers will have techniques to help improve QoS assurance of large-scale distributed systems.

**Paper organization.** The remainder of this paper is organized as follows: Section 2 introduces a case study of a representative large-scale distributed system; Section 3 defines and discusses challenges associated with unit testing QoS attributes; and Section 4 provides concluding remarks.

## 2 Case Study: The QED Project

The QED project [1] is a multi-organization collaboration designed to improve the *Global Information Grid (GIG)* [2] middleware so it can meet QoS requirements of end-users applications and different operational scenarios. The QED project has been used in prior research, such as unit testing performance [5] of large-scale component-based distributed. Figure 1 illustrates QED in the context of the GIG.

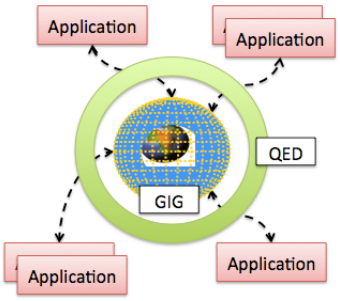


**Figure 1. QED and the GIG middleware**

To briefly reiterate an overview of the QED project, its aim is to provide reliable and real-time communication middleware that is resilient to the dynamically changing conditions of the GIG environment. The QED project is also scheduled to run for several more years. Since applications that will utilize the GIG middleware, and in turn QED, will not be available for several years later, QED developers are using the CUTS [4] system execution modeling tool perform integration testing during early stages of the software lifecycle and continuously.

QED developers use CUTS by (1) modeling behavior and workload of applications under development that will use the GIG and QED middleware; (2) synthesizing a complete test system from constructed models that target the GIG middleware; and (3) executing the synthesized test system on its target architecture and in a representative test environment. This enables QED developers to evaluate the improvements QED middleware and its features is providing GIG middleware during early stages of the software lifecycle instead of waiting until complete system integration, *i.e.*, late in the software lifecycle.

In prior research [5], UNITE was developed as a technique and tool for unit testing QoS attributes. In that research effort, performance of the GIG was the only QoS attribute evaluated. The QED middleware, however, has to ensure multiple QoS attributes, such as reliability and security, when scaling to meet the demands of end-user applications and operational scenarios. Applying UNITE to evaluate other QoS attributes, however, revealed the following limitations in UNITE and CUTS:

- **Limitation 1: Unclear definition of unit testing QoS attributes.** The meaning of unit testing functional attributes is well-understood by the QED developers. Little research, however, has investigated the meaning of unit testing QoS attributes. QED developers therefore are unsure of what it actually means to unit test multiple QoS attributes, *i.e.*, not only performance, in comparison to unit testing functional attributes.

- **Limitation 2: Determining success/failure of QoS unit test.** When evaluating functional attributes of a large-scale distributed system, its relatively simple the determine success/failure of a functional unit test. Success/failure of a QoS unit test, however, depends on many factors, such as operating environment and configuration of the large-scale distributed system. QED developers therefore need better methodologies for determining the success/failure of QoS unit tests.

- **Limitation 3: Highlighting multiple views of a QoS unit test.** A single QoS unit test is used to evaluate a single QoS attribute. QoS unit test, however, are evaluated based on metrics collected while the software system is executing in its target environment. This implies that evaluation of each QoS attribute requires its own test run. This approach can be an expensive and time-consuming process. QED developers therefore need better techniques for extracting different analysis (or multiple data views) from a QoS unit test.

Due to these limitations, it is hard to QED developers to

unit test multiple QoS attributes continuously throughout the software lifecycle. The remainder of this paper discusses the aforementioned limitations in more detail and challenges associated with each limitation.

# 3 Challenges Associated with Unit Testing QoS Attributes

This section discusses challenges associated with unit testing QoS attributes continuously throughout the software lifecycle.

## 3.1 Definition of QoS Unit Testing

**Overview.** Unit testing functional attributes of large-scale distributed systems is a well-defined process. It traditionally involves testing functional attributes, such as build and behavior correctness, for small portions of the system, such as a method on a class or an input port of a component, on its target architecture. Unit testing QoS attributes should have similar meaning as unit testing functional attributes of large-scale distributed systems. Unit testing QoS attributes, however, must be performed on both the target architecture and in a representative environment that will produce realistic results.

**Challenges.** Although unit testing QoS attributes should have similar meaning as unit testing functional attributes, achieving similarity is hard. This is due in part to the fact that side-effects experienced during functional and QoS testing are different. For example, success/failure of a class's method will dictate success/failure of a component that utilizes that class's method within its application-logic during functional unit testing. In contrast, success/failure in performance of a class's method *may not* determine success/failure in performance for a component that utilizes that class's method.

Because QoS test results are not as deterministic when compared to functional test results, it is hard define QoS unit testing in similar terms as functional unit testing. Moreover, QoS attributes are traditionally systemic attributes, such as end-to-end response time, in large-scale distributed systems. The granularity of QoS unit test is therefore another challenge because when compared to functional unit testing, QoS unit testing is a more expensive and time-consuming, which is impacted by test granularity.

## 3.2 Success or Failure of a QoS Unit Test

**Overview.** Section 3.1 discussed how the success/failure of a QoS unit test do not have similar effects as the success/failure of a functional unit test. In order to increase the level of confidence in large-scale distributed system QoS attributes, however, determining the success/failure of a QoS unit test is critical because it helps guide development efforts. Distributed system developers, therefore, need improved methodologies that will help them determine the success/failure of a QoS unit test.

**Challenges.** In contrast to functional unit testing where success/failure a functional unit test has a more deterministic effect on success/failure another functional unit, QoS attributes do not have as deterministic effect on other QoS attributes. For example, increasing the security of a class or a single component, may have a negative impact on performance. Likewise, increasing the fault tolerance of a large-scale distributed system may have a negative impact on the systems security because there are more replicas to secure. Moreover, the result of a QoS unit test depends heavily on the target execution environment, which must be as realistic as possible. For example, executing a QoS unit test on a development machine will yield different results than executing the same QoS unit test on a production machine.

In functional unit testing, *test oracles* [3], which are mechanisms that evaluate system requirements against tests, determine the success/failure of functional unit tests. Test oracles are traditionally extracted from a large-scale distributed system documentation [10]. In many cases, it is easier to derive test oracles for functional unit tests because because functional requirements are a primary concerns and present in specification and requirement documents. In contrast, QoS attributes are taken into consideration after a system is completely functional and not meeting vague QoS requirements, such as "the system must have high availability, performance, and security" [9]—similar to the QED project for the GIG middleware.

Although test oracles will help determine success/failure of a QoS unit test, they will not have the same behavior as test oracles for functional unit tests. In QoS testing, success/failure is not as clear as it is in functional unit testing. Moreover, when dealing with multiple levels of granularity in QoS unit testing, as discussed in Section 3.1, test oracles will need to have intelligent reasoning capabilities to cipher through results from multiple levels of granularity when determining success/failure. Otherwise, distributed system developers will waste time and effort correcting false-negative results and not be made aware problems masked by false-positive results.

## 3.3 Satisfying Multiple Views Simultaneously

**Overview.** Evaluating a QoS unit test for large-scale distributed systems requires executing the system on its target architecture and in its target environment. This enables the distributed system developer to obtain realistic results and feedback and help guide their development efforts. Evaluating a single QoS attribute via a QoS unit test, however, is a time-consuming task. For example, duration of an QoS unit

test that evaluates performance and reliability of a large-scale distributed system is not known *a priori*. Distributed system developers therefore needed improved capabilities to evaluate multiple QoS attributes (or views) simultaneously to reduce this complexity of QoS unit testing.

**Challenges.** Evaluating a QoS unit test for a large-scale distributed system requires instrumenting the system to collect the necessary metrics for the evaluation. Instrumenting a large-scale distributed system, however, can impact collected metrics and different QoS attributes. For example, the end-to-end response of a large-scaled distributed system will be greater when more metrics are collected than when no metrics are collected. Moreover, instrumenting a large-scale distributed system to collect metrics for to evaluating multiple QoS attributes can have dire effects on the results because it requires collecting more metrics than what is needed to evaluate a single QoS attribute.

Once metrics have been collected for a QoS unit test, the QoS unit must be evaluated. UNITE is capable of evaluating a single QoS unit test for a single QoS attribute. This, however, implies that distributed system developers that use UNITE must execute a large-scale distributed system many times to evaluate different QoS attributes. Moreover, different distributed system developers may want to analyze different views of the metrics, such as a subset of the metrics occurring between certain time intervals or different QoS attributes from a single execution of the system. Techniques for data mining and analyzing metrics collected for a QoS unit test is therefore another challenge in reducing complexity and increasing capabilities of QoS unit testing.

## 4 Concluding Remarks

As large-scale distributed system increase in size and complexity, ensuring their QoS attributes continuously throughout the software lifecycle is becoming more critical. This will help reduce the amount of time, money, and effort exerted late in the software lifecycle, *i.e.*, at system integration time, rectifying problems related to not satisfying QoS attributes. This paper discussed some of the challenges related to enabling QoS unit testing, which is aimed at increase QoS assurance—similar to how functional unit testing helps increase quality assurance. By addressing the challenges discussed in this paper and improving QoS unit testing capabilities, large-scale distributed system developers will have the necessary tools and techniques to build larger and more complex distributed systems where QoS attributes are not an afterthought.

## References


[1] BBN Technologies Awarded $2.8 Million in AFRL Funding to Develop System to Link Every Warfighter to Global Information Grid. BBN Technologies—Press Releases, www.bbn.com/news_and_events/press_releases/2008_press_releases/pr_21208_qed.

[2] Global Information Grid. The National Security Agency, www.nsa.gov/ia/industry/gig.cfm?MenuID=10.3.2.2.

[3] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001.

[4] J. H. Hill, D. C. Schmidt, and J. Slaby. Evaluating Quality of Service for Enterprise Distributed Real-time and Embedded Systems. In P. F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*. Idea Group, 2007.

[5] J. H. Hill, H. A. Turner, J. R. Edmondson, and D. C. Schmidt. Unit Testing Non-functional Concerns of Component-based Distributed Systems. In *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*, Denver, Colorado, Apr. 2009.

[6] A. Hunt and D. Thomas. *Pragmatic Unit Testing in C# with NUnit*. The Pragmatic Programmers, Raleigh, NC, USA, 2004.

[7] S. E. Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, Jun 2006.

[8] V. Massol and T. Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[9] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[10] D. Peters. *Generating a Test Oracle From Program Documentation*. PhD thesis, McMaster University, Hamilton, Ontario L8S4L8, 1995.

[11] Rittel, H. and Webber, M. Dilemmas in a General Theory of Planning. *Policy Sciences*, pages 155–169, 1973.

[12] C. Smith and L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, Boston, MA, USA, September 2001.