Evaluating Distributed Real-time and Embedded System Test Correctness using System Execution Traces

James H. Hill
Dept. of Computer and Information Science
Indiana Univ.-Purdue Univ. Indianapolis
Indianapolis, IN USA
hillj@cs.iupui.edu

Pooja Varshneya and Douglas C. Schmidt
Dept. of Elec. Engineering and Comp. Science
Vanderbilt University
Nashville, TN USA
{p.varshneya, d.schmidt}@vanderbilt.edu

ABSTRACT

Effective validation of enterprise distributed real-time and embedded (DRE) system quality-of-service (QoS) properties requires evaluating system capabilities in their target execution environments. This validation process traditionally involves executing DRE systems composed of many software components on many hardware components connected via networks. Evaluating the correctness of such tests is hard, however, since it requires validating many system states dispersed across many hardware/software components.

This paper provides two contributions to research on validating DRE system capabilities and QoS properties. First, it presents the *Test Execution (TE) Score*, which a methodology for validating execution correctness of enterprise DRE system tests. Second, it empirically evaluates the TE Score in the context of a representative enterprise DRE system. The results of this evaluation show that the TE Score method can determine the percentage correctness in test execution—thereby increasing confidence in QoS assurance—and help DRE system developers improve test quality.

1. INTRODUCTION

Enterprise distributed real-time and embedded (DRE) systems, such as large-scale traffic management systems, manufacturing and control systems, and global financial systems, are increasing in *size* (*e.g.*, number of lines of source code and number of hardware/-software resources) and *complexity* (*e.g.*, envisioned operational scenarios and target execution environments) [11]. It is therefore critical to validate their quality-of-service (QoS) properties (such as event prioritization, latency, and throughput) in their target environments continuously throughout their software lifecycles. Continuous validation enables DRE system testers to locate and rectify performance bottlenecks with less time and effort than deferring validation to final system integration [18, 28].

System execution modeling (SEM) [20, 27] is a promising approach for continuously validating QoS properties of DRE systems in their target environment throughout the software lifecycle. In particular, DRE system testers can use SEM tools to:

1. Model DRE system behavior and workload at high-levels of abstraction using domain-specific modeling languages [14],
2. Synthesize complete test systems from constructed behavior and workload models that conform to the DRE system's target environment (*e.g.*, middleware platforms, operating systems, and networks) similar to rapid prototyping [3], and
3. Execute the test system in its target execution environment using emulation techniques to validate QoS properties.

SEM tools also allow DRE system testers to replace emulated components of DRE systems with their actual counterparts as the development process unfolds. This incremental development and testing method can yield more realistic results, thereby increasing confidence in the QoS assurance process.

Although SEM tools can help validate QoS properties in representative target environments, conventional SEM tools do not ensure that QoS tests themselves execute correctly. For example, it is possible for a DRE system to execute incorrectly due to *transient errors* even though the test *appeared* to execute correctly [29], *e.g.*, since it did not detect the effects of node failures on QoS properties because injected failures did not occur as expected. Likewise enterprise DRE systems have many competing and conflicting QoS properties that must be validated [11]. For example, end-to-end response time may meet specified QoS requirements, but latencies between individual components may not meet specified QoS requirements due to software/hardware contention and QoS trade-off requirements, such as prioritizing system reliability over intermittent response time.

These problems are exacerbated when metrics needed to validate these concerns are dispersed across many hardware/software components. Developers of DRE systems currently determine test execution correctness via conventional techniques, such as manually inserting checkpoints and assertions [5]. Unfortunately, these techniques can alter test behavior and performance, are locality constrained, and focus on functional concerns rather than QoS properties. DRE system testers therefore need improved techniques that help reduce the complexity of ensuring test correctness when validating enterprise DRE system QoS properties in their target environments.

Solution approach → Correctness validation via system execution traces. System execution traces [4] are artifacts of executing a software system (*e.g.*, an enterprise DRE system) in its target environment. These traces log messages that capture system state during different execution phases, such as component activation versus passivation. System execution traces can also capture metrics for validating test execution correctness, such as event timestamps that determine which components in an end-to-end activity exceeded its allotted execution time. In the context of correctness validation, system execution traces can be used to quantify the correctness of a enterprise DRE system test that validates QoS properties in terms of its states and trade-off analysis of such properties.

This paper describes a method called the *Test Execution (TE) Score*, which uses system execution traces to validate enterprise DRE system test correctness. DRE system testers use the TE Score by first defining valid and invalid DRE system states and QoS properties, such as number of events processed or acceptable response time(s) for an event. The TE Score then uses system execution traces to evaluate test correctness using the specified (in)valid state and QoS properties. Our experiments show how applying the TE Score to a representative enterprise DRE system can provide DRE system testers with a correctness grade (*i.e.*, a percentage) that quantifies how well their tests execute. Moreover, the TE Score helps identify test errors that must be rectified to improve correctness and increase confidence levels in QoS assurance for enterprise DRE systems.

Paper organization. Section 2 introduces a representative enterprise DRE system case study to motivate the need for the TE Score; Section 3 describes the design and implementation of the TE Score; Section 4 presents results of experiments that applied the TE Score to the case study; Section 5 compares the TE Score with related work; and Section 6 provides concluding remarks.

2. CASE STUDY: THE QED PROJECT

The *QoS-Enabled Dissemination* (QED) [17] project is information management middleware designed to meet QoS requirements of component-based enterprise DRE systems in the Global Information Grid (GIG) [1], which is a large-scale distributed system [11] designed to ensure that different applications can collaborate effectively and deliver appropriate information to users in a timely, dependable, and secure manner. QED's aims to provide reliable and real-time communication middleware that is resilient to the dynamically changing conditions of GIG environments. Figure 1 shows QED and an example in the context of the GIG.

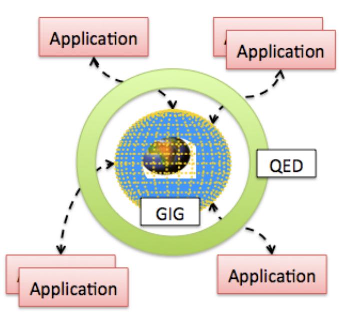

Figure 1: QED Relationship to the GIG

The QED project has been used in previous studies related to distributed system testing, such as unit testing QoS properties [10]. The project is now in its second year of development and is slated to run for several more years. Since the QED middleware is infrastructure software, applications that use it cannot be developed until the middleware itself is sufficiently mature. The QED project thus faces a common software engineering problem: the *serialized-phasing development problem* [25], where systems are developed in different layers and phases throughout their lifecycle. In this software development model, design flaws that negatively impact QoS properties are often not identified until late in the software lifecycle, *e.g.*, system integration time, when it is much more expensive to resolve the flaws [18, 28].

To overcome the serialized-phasing development problem, QED testers use SEM tools to validate QoS properties of the QED middleware on the target architecture continuously throughout the software lifecycle. In particular, they use CUTS and UNITE (see Sidebar 1) to ensure QED's enhancements to the GIG middleware actually improve QoS properties of the existing GIG middleware. QED testers use (1) CUTS to measure the extent to which QED capabilities can handle to the dynamically changing conditions of the GIG and (2) UNITE to validate DRE system QoS properties irrespective of system composition (*i.e.*, how the system is structured and what components communicate with each other) and implementation detail (*i.e.*, target architecture, language, and technology).

Sidebar 1: Overview of CUTS and UNITE

The *Component Utilization Test System* (CUTS) [9] is a system execution modeling tool for large-scale distributed real-time and embedded (DRE) systems that enables DRE system testers to conduct system integration test and validate QoS properties on the target architecture during early phases of the software lifecycle. DRE system testers use CUTS via the following steps:

1. Use domain-specific modeling languages [14] to model behavior and workload at high-levels of abstraction;
2. Use generative programming techniques [6] to synthesize a complete test system for the target architecture; and
3. Use emulation techniques to execute the synthesized system on its target architecture and validate its QoS properties, such as end-to-end response time, latency, and scalability, in its target execution environment.

DRE system testers can also replace emulated portions of the system with its actual counterparts as their development completes. This incremental approach enables DRE system testers to perform *continuous system integration testing*, *i.e.*, the process of execution system integration test to validate QoS properties continuously throughout the software lifecycle. CUTS supports validating QoS properties on several network communication architectures, such as the GIG/QED middleware, CIAO [12], OpenSplice [22], RTI-DDS [23], and TCP/IP.

The *Understanding Non-functional Intentions via Testing and Experimentation* (UNITE) [10] tool distributed with CUTS enables DRE system testers to generate QoS performance graphs from system execution traces and validate QoS properties. DRE system testers first generate system execution traces by execution the system in its target environment. They then use UNITE to define a dataflow model [7] and use it to mine metrics of interest (*e.g.*, event throughput) for a QoS property from the system execution trace. Finally, UNITE constructs a QoS performance graph that shows that QoS property's data trend throughout the lifetime of the system (*i.e.*, how that QoS property changed with respect to time). Section 3.1 presents a more detailed overview of UNITE's methodology.

Although CUTS and UNITE enabled QED testers to evaluate QED's enhancements to QoS properties of GIG middleware, they still face the following challenges that make it hard to effectively apply CUTS and UNITE to the QED project.[1]

Challenge 1: Inability to validate execution correctness of QoS test. Executing an enterprise DRE system QoS test requires running the system in its target environment. This environment consists of many hardware/software components that must coordinate with each other. To determine the correctness of QoS test execution, DRE system testers must ensure that all hardware/software resources in the distributed environment behave correctly.

QED testers therefore need a method that simplifies validating if QoS validation tests execute correctly. This method should automate the validation process so testers need not manually check all hardware/software resources for correctness. The validation method should also minimize false negatives (*e.g.*, stating the QoS test executes correctly, but in reality it failed to meet different QoS requirements, such as sending the correct number of events within a given time period). Addressing this challenge enables QED testers can have greater confidence levels in QoS assurance. Section 3.1 describes how the TE Score method addresses this problem using state-based specifications.

Challenge 2: Inability to perform trade-off analysis between QoS properties. QoS properties are a multi-dimension concern [16]. It is hard to simultaneously ensure all enterprise DRE system QoS properties with optimal performance, such as ensuring high reliability and low end-to-end response time; high scalability and high fault tolerance; or high security and low latencies. Resolving this challenge requires trade-off analysis that prioritizes what QoS properties to validate (or ensure) since some are more important than others. For example, QED testers must ensure that high priority events have lower latency than lower priority events when applying QED's enhancements to GIG middleware.

After QED testers validate the correctness of QoS test execution (*i.e.*, resolve challenge 1), they ideally want to validate multiple QoS properties simultaneously since it is time-consuming to validate a single QoS property in isolation. QED testers therefore need a method that will assist in this trade-off analysis between multiple dimensions of QoS properties. Moreover, the methodology should allow QED testers to determine (1) what QoS properties are most important, (2) quantify the correctness of QoS test execution based on the specified priorities, and (3) help identify and prioritize where improvements in QoS test execution are needed. Section 3.2 describes how the TE Score method addresses this challenge by adding priorities and weights to the state-based specifications used to validate test correctness.

3. THE DESIGN AND IMPLEMENTATION OF THE TE SCORE

This section describes the design and implementation of the TE Score, which uses system execution traces to validate QoS test correctness test. Examples from the QED case study introduced in Section 2 are used throughout this section to show how the TE Score can be applied to a representative enterprise DRE system.

3.1 Specifying QoS Test Execution States

Validating QoS test correctness requires evaluating an enterprise DRE system's state and QoS properties over its complete lifetime, *i.e.*, from the time the system is deployed to the time it shuts down. This requirement is necessary because QoS properties, such as latency and end-to-end response time, often fluctuate over the lifetime of an enterprise DRE system due to hardware/software contention, component failures, and workload variation. There can also be states that the enterprise DRE system must reach (and maintain) to properly evaluating QoS properties, including ensuring a component has received the correct number of events within a time period to ensure end-to-end response time is evaluated under the expected workload.

Our prior work [10] has used system execution traces to capture both system state and metrics for evaluating QoS properties. Since system execution traces are a collection of text-based messages, this method provides an architecture-, technology-, and language-independent mechanism for capturing state and QoS metrics. The challenge, however, is mining the system execution traces and extracting the important information for validating correctness in QoS test execution, as described in Challenge 1.

Our prior work [10] has also shown how to mine system execution traces and extract data and metrics of interest using dataflow models. In the context of testing enterprise DRE systems, a dataflow model $DM = (LF, CR)$ is defined as:

- A set LF of log formats that have a set V of variables identifying what data to extract from log messages in a system execution traces. These log formats will identify many occurrences of the same message in a system execution trace where their difference is captured in V.
- A set CR of causal relations that specify the order of occurrence for each log format such that $CR_{i,j}$ means $LF_i \rightarrow LF_j$, or LF_i occurs before LF_j [26]. These relations help determine how data flows across different application contexts, such as one component sending an event to another component deployed on a different host.

Although our prior work has used dataflow models to validate QoS properties, these models can also be leveraged to validate QoS test execution states. In particular, the set of variables V across the set of log formats LF in DM capture the state and metrics of an enterprise DRE system at any given point in time throughout its lifetime. This information can thus be used to validate the correctness of test execution.

Defining QoS execution states. In the context of validating test execution correctness an execution state $s = (DM, V', P)$ of an enterprise DRE system is formally defined as:

- A dataflow model DM that contains a set of variables for capturing the system's state and metrics throughout its execution lifetime;
- A set of variables V' used to evaluate the system's state of interest where $V' \in V$ in the dataflow model DM; and
- A preposition P that captures the context and expected value the context over the of variables v such that $C_v \rightarrow E_v$ means C_v defines the context for the expected value of P (*e.g.*, the name of the component) and E_v defines the actual value (or effect) for the given context (*e.g.*, the response time for the specified component).

[1] Although these challenges are motivated in the context of QED, they apply to other enterprise DRE systems that must validate their QoS tests.

Implementing QoS test execution states in TE Score. To realize test execution states in TE Score, we leverage UNITE's capabilities (see Sidebar 1) for specifying dataflow models. In particular, QED testers create a dataflow model in UNITE, as shown in Figure 2. Using the set of variables defined in the dataflow model, QED testers then select variables to define an execution state that can help validate the execution correctness for their QoS test. This execution state is specified as a preposition P where the context C_v is an expression for defining the scope of the evaluation and the expected value E_v is the expected state for the specified context.

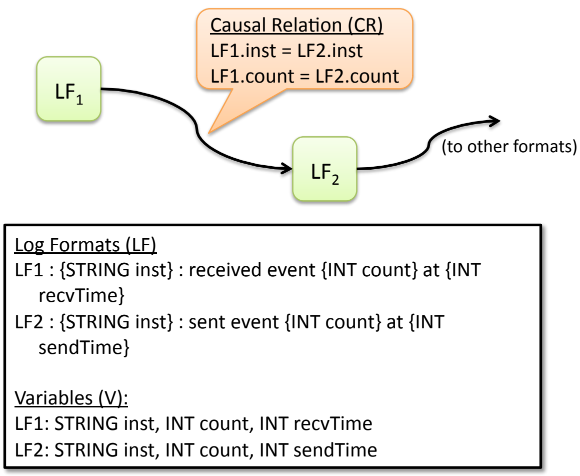

Figure 2: Example of a Dataflow Model in UNITE

Listing 1 shows a TE Score example that validates whether response time for the `ConfigOp` component is *always* less than 30 msec. In this example QED testers select `LF1.instName`, `LF1.recvTime`, and `LF2.sendTime` from the dataflow model in Figure 2 and then define a preposition where the context checks the value captured in the variable `LF1.instName`. If the specified context is valid, *i.e.*, C_v is `true`, then the expected value of the context is evaluated, *i.e.*, E_v is tested. The execution state is ignored if the context is invalid.

```
Cv: LF1.instName = ''ConfigOp''
Ev: (LF2.sendTime − LF1.recvTime) < 30
```

Listing 1: Example State Specification in the TE Score

3.2 Specifying Execution Correctness Tests

Section 3.1 showed how the TE Score leverages dataflow models to capture and define a single execution state for validating execution correctness for a QoS test. In practice, however, there can be many execution states that must be evaluated to determine execution correctness for a QoS test. For example, a DRE system tester may need to validate that response-time of a single component is less than 30 msec, the arrival rate of events into the system is 10 Hz, and the end-to-end response time for a critical path of execution is less than 100 msec because under those conditions do their experiments produce meaningful workload for validating QoS properties. Likewise, it may be necessary to ensure the test execution does not reach an invalid state because it may be easier to express QoS test execution states in terms of its invalid states, as opposed to its valid states.

It is often hard, however, to validate all the execution states of a QoS tests due to the enormous number of plausible states the system can enter. Using the previous example, for instance, it can be hard for DRE system testers to ensure that the response time of a single component is less than 30 msec and the end-to-end response time is less than 100 msec because QoS properties traditionally conflict with each other. Due to conflicting interests when specifying execution states, therefore, it is often necessary to conduct trade-off analysis for the different execution states. For example, DRE system testers may want to specify that ensuring end-to-end response time is more important than ensuring the response time of a single component, as described in Challenge 2.

Defining execution correctness test specifications. Since an execution state s can be either invalid or valid—and trade-off capabilities are needed to validate correctness of QoS test execution—we must extend the definition of an execution state s such that $s' = (s, t, p, min, max)$ where:

- s is the original QoS test execution state (see Section 3.1);
- t is the QoS execution test state type (*i.e.*, either invalid or valid); and
- p is the priority (or importance) of the QoS test execution state, such that lower priorities are considered more important. This priority model is used because it does not put a predefined upper bound on priorities and offers a more flexible approach when assigning priorities to QoS test execution states;
- min is the minimum number of occurrences the specified state s can occur throughout the execution of the QoS test since QoS value can fluctuate and are rarely constant; and
- max is the maximum number of occurrences that the specified state s can occur throughout the execution of the QoS test since QoS values can fluctuate and are rarely constant;

Using the new definition of a QoS test execution state s', it is now possible to define a correctness test $CT = (S)$ as:

- A set S of QoS execution test states s' where each state $s' \in S$ determines the execution correctness for a QoS test.

Implementing execution correctness test specifications in TE Score. To realize execution correctness test specifications in TE Score, QED testers specify a set of states that determine the execution correctness for a QoS test. Each state is defined as either valid (*i.e.*, an allowable state) or invalid (*i.e.*, a state that is not allowed). Likewise, each state in the correctness test is given a priority that determines its level of importance when conducting trade-off analysis between different execution states.

s'_1 : $LF3.instName = \text{“}Receiver''$
$\rightarrow \frac{LF4.eventCount}{LF5.stopTime - LF6.startTime} > 5$
t = `valid`
p = 2
min = 1
max = 1

$s'_2:\ LF1.instName = \text{“}ConfigOp''$
$\rightarrow (LF2.sendTime - LF1.recvTime) > 30$
t = `invalid`
p = 4
min = 1
max = unbounded

$s'_3:\ LF7.endTime - LF8.beginTime < 100$
t = `valid`
p = 1
min = 0
max = unbounded

Listing 2: Example Correctness Test Specification in the TE Score

Listing 2 shows an example of a correctness test that validates the correctness of QoS test execution. As shown in this example, there are three different QoS execution states:

1. A state validating the arrival rate of events into a component named `Receiver` is 5 events per second, which should only occur once during the execution of the QoS test;
2. A state validating that the response time for a component named `ConfigOp` is never greater than 30 msec throughout the execution of the QoS test; and
3. A state validating the end-to-end response time of an event that is always less than 100 msec throughout the entire execution of the QoS test.

3.3 Evaluating Execution Correctness Specifications

After defining an execution correctness specification test (see Section 3.2), the final process is evaluating it. The main goal of the evaluation process is to provide a metric that quantifies the degree to which a QoS test executes correctly based on its specified states. This metric serves a two purposes: (1) it helps DRE system testers determine how well their tests execute and meet expectations, *e.g.*, ensuring a component receives the correct number of events within a time period while meeting end-to-end response times, and (2) it identifies areas where tests may need improvement, *e.g.*, possibly reducing the arrival rate of an event because its negatively impacting latencies and system scalability.

Given the definition of an execution correctness specification for a QoS tests presented in Section 3.2, evaluating these tests must account for both the QoS execution state's type (*i.e.*, invalid or valid) and priority. Failure to account for these two properties can yield false negative results that do not provide meaningful information to DRE system testers. For example, the higher priority states (*i.e.*, states with a lower priority number) should have a greater weight on the overall evaluation of an execution correctness specification. Likewise, the overall evaluation of an execution correctness specification should be negatively impacted whenever a valid execution state is not reached, or an invalid execution state is reached.

It is therefore possible to use the priorities (or weights) and state types to derive a weighted grading system such that Equation 1 calculates the weight of a given state s' in a correctness specification:

$$weight(s') = maxprio(S) - prio(s') + 1 \tag{1}$$

As shown in Equation 1, $maxprio(S)$ determines the highest priority number for a QoS execution state (*i.e.*, the QoS execution state with the least priority) and $prio(s')$ is the priority value of the specified QoS execution state.

Since a weighted grading system is used to validate the correctness of QoS tests execution, QoS execution states of greater importance will have more influence on the overall grade than QoS execution states of less importance. In particular, obtaining valid states increases the QoS test execution correctness grade and reaching invalid states decreases the grade. Likewise, if a valid state is not reached, then the grade is decreased, whereas and if an invalid state is not reached the the execution correctness grade is increased.

Equations 2 and 3 help determine the number of points representing valid QoS execution states that can be reached or invalid QoS execution states that cannot be reached for a given dataset DS.

$$points(DS, s') = \begin{cases} evaluate(DS, s') = true & weight(s') \\ evaluate(DS, s') = false & 0 \end{cases} \tag{2}$$

$$points(DS, S) = \sum_{s' \in S} points(DS, s') \tag{3}$$

Equation 5 and Equation 4 also shows the equation that determines the final grade for execution correctness specification of a QoS test.

$$maxpoints(S) = \sum_{s' \in S} weight(s') \tag{4}$$

$$G(DS, S) = \frac{points(DS, S)}{maxpoints(S)} \times 100 \tag{5}$$

As highlighted in this equation, the final grade for execution correctness for a given dataset is determined by number of points award for each test execution state, *i.e.*, number of valid states reached and invalid states not reached, divided by the number of total possible points.

Implementing correctness test evaluation in the TE Score. To realize correctness test evaluation in the TE Score, the TE Score leverages UNITE's capabilities for data mining system execution traces and constructing a dataset that represents the given dataflow model. Once the dataset is constructed, TE Score processes each state in the correctness test to derive a grade for the test. Algorithm 1 shows the general algorithm TE Score uses to evaluate a QoS execution state when grading the execution correctness of a QoS test using Equations 2 – 5.

As shown in the Algorithm 1, given the dataset DS constructed by UNITE and the execution state s', the SQL state for the state is constructed (line 6). After constructing the SQL statement, the SQL statement is applied to the dataset and the number of rows in the result set in stored. If the state s' is a valid state, and the number of rows is less than the min occurrences or greater than the max occurrences, then 0 points is returned (line 11). Likewise, if the state s' is an invalid state and the count falls within the specified range [min, max], then 0 points is returned (line 15).

Handling false negatives. There can be cases when querying the dataset for the specified state may yield false negatives. For example, if the expected value E_v of the context C_v is true, that does

Algorithm 1 General algorithm for evaluating execution correctness state.

```
procedure EVALUATE(DS, s′, P)
    DS: dataset from UNITE
    s′: execution state
    P: max points

    sqlstr ← sqlstmt(s′)
    n ← get_count(DS, sqlstr)

    if is_valid(s′) then
        if n ≤ min(s′) ∨ n ≥ max(s′) then
            return 0
        end if
    else
        if n >= min(s′) ∧ ≤ max(s′) then
            return 0
        end if
    end if

    if is_unbounded(s′) ∧ has_false_negs(DS, s′) then
        return 0
    end if

    return points(s′, P)
end procedure
```

not necessarily mean that E_v is never false because the query's result only returns data that matches the specified query. The query does not validate if the dataset contains data that invalidates the expected value (*i.e.*, check for false negatives), which can occur when the max value is *unbounded* (*i.e.*, the state can be reached infinite number of times).

To prevent false negatives from occurring in the evaluation, the TE Score negates the expected value E_v and applies it to the dataset. If the number of rows in the new result set is greater than the $min(s')$, then the state has a false negative. For example, assume the bounds of occurrence for s' is [0, unbounded], which means that state should always occur. Therefore when evaluating the negation of the expected value E_v, the result set should be empty, *i.e.*, the row count of the new result set should be 0.

Determining the final score. Once all points for the correctness tests are accumulated, the final step is assigning a grade to the test. This grade helps distributed system testers determine how well their tests are executing. Moreover, it helps identify what QoS execution states are candidate for resolving and improving results in QoS assurance. The TE Score therefore uses Equation 5 to assign a final grade to the correctness test by dividing the accumulated points by the total number of points possible, then multiplying that result by 100 to get a percentage.

4. APPLYING THE TE SCORE TO THE QED PROJECT

This section presents results from applying the TE Score to the QED case study introduced in Section 2 to evaluate the test correctness of various experiments performed on the QED middleware.

4.1 Experiment Setup

As discussed in Section 2, the QED project aims to enhance QoS concerns of GIG middleware by adding adaptive information management capabilities to it. Enhancing the GIG middleware in this manner enables it to prioritize user requests and meet QoS requirements of mission-critical applications in resource-constrained environments. To ensure that the QED project does in fact improve the QoS capabilities of GIG middleware, QED testers have constructed several experiments designed to evaluate the QED enhancements. In particular, QED testers wanted to evaluate the following QED capabilities:

- **Prioritized services for high priority users.** Experiments were performed to verify that higher importance subscribers received prioritized services, *e.g.*, higher throughput and low response time for receiving information objects, when compared to lower importance subscribers.
- **Adaptive resource management capabilities to ensure subscriber QoS requirements.** Experiments were designed to simulate resource constrained environments, such as limited dissemination bandwidth or high CPU utilization conditions, to validate the adaptive capabilities of QED middleware and ensure subscriber QoS properties were not degraded when compared to lower importance subscribers under such conditions.
- **QoS performance measurement for higher importance versus lower importance subscribers.** Experiments were performed to measure QoS properties of higher importance versus lower importance subscribers and validate that the QED/GIG middleware met its minimum QoS requirements for all subscribers.

QED testers used CUTS and UNITE to design several experiments that evaluated these concerns empirically. These experiments contained many software components running on many hardware components communicating via a shared network. The application components for their experiments were first modeled using CUTS's behavior and workload domain-specific modeling languages [14]. The constructed models were then used to generate source code for the QED/GIG middleware. Finally, UNITE was used to mine metrics of interest from system execution traces and generate performance graphs that illustrated a given QoS metric's data trend (*i.e.*, how the metric changes with respect to time.

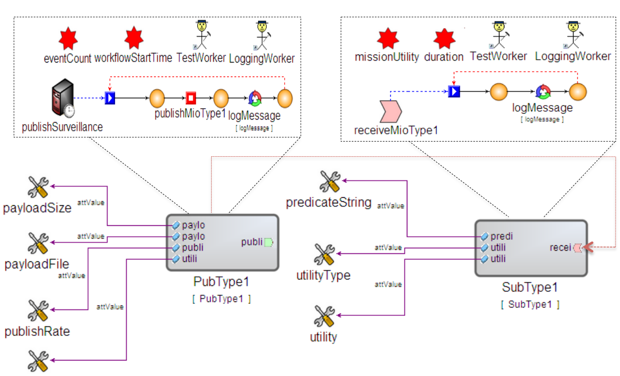

Figure 3: QED Clients Modeled Using CUTS

Figure 3 highlights an example CUTS model for an experiment created by QED testers. The workload generators in this figure were used to simulate publisher (PubType1) and subscriber (SubType1) client behavior. Publisher clients have attributes (such as payload size and publish rate) that can be modified at runtime to change publisher behavior. Likewise, subscriber clients have attributes (such as information matching predicates) that can be modified at runtime to alter the types of information objects the subscriber can receive during the execution of an experiment. `TestWorker` and `LoggingWorker` element in Figure 3 represent workload generators that capture system execution traces for either the publisher and subscriber clients. These system execution traces are used by UNITE to create datasets based on dataflow models and validate QoS properties, such as throughput and latency.

Although QED testers used CUTS and UNITE to reduce complexities (such as experiment design/implementation, data collection, and data analysis) associated with validating QED's enhancements to the GIG middleware, QED testers were still faced with the challenge of ensuring that their experiments executed correctly, *e.g.*, setting experiment configurations and ensuring that none of the publishers, subscribers, and QED server fail at runtime. If any configuration errors or runtime failures occur on an experiment nodes or the experiment fails to show expected system behavior, QED testers originally had to troubleshoot the issue manually by mining dense system execution traces for the appropriate error messages.

QED testers therefore decided to use the TE Score to evaluate the execution correctness of their experiments. More specifically, the QED testers leveraged the generated system execution traces used by UNITE to validate different execution states of QED experiments (see Section 3.1 and Section 3.2) and grade the quality of each individual experiment based on its execution states (see Section 3.3). Listing 3 highlights example log formats used to mine the generated system execution trace and example states for validating and grading the execution correctness of several QED experiments using the TE Score.

Log Formats

```
LF1: {STRING client} started with
     environment {STRING env}
LF2: {STRING client} started at {LONG
     startTime} with publishrate= {INT rate}Hz
LF3: {STRING client} ended at {LONG endTime}
LF4: {STRING client} received {LONG evid}
     with payload of size {INT size} bytes
```

QoS Test Execution States

(a) Environment initialization state

s_1: LF1.env = "qed"
t_1: valid
p_1: 1

(b) Publisher initialization state

s_2: LF2.client = "pub1"and LF2.rate = 6
t_2: valid
p_2: 1

c). Publisher runtime execution states

s_3: LF2.client = LF3.client and
LF3.endTime - LF2.startTime=360000
t_3: valid
p_3: 1

(d) Subscriber runtime execution states

s_4: LF4.size = 1024
t_4: valid
p_4: 1

s_5: LF4.client = "sub1"and LF4.evid > 5400
and LF4.evid < 6480
and LF4.size = 1024
t_5: valid
p_5: 2

s_6: LF4.client = "sub3" and
(LF4.evid > 150 or LF4.evid < 50)
and LF4.size = 1024
t_6: not valid
p_6: 4

Listing 3: Examples of the TE Score States

As shown in Listing 3, LF1, ..., LF4 define log messages used to capture clients' states from the beginning to the end of the experiment, and $s_1, \ldots, s_6$ define test execution states that occur during the experiment. The LF1 and LF2 log formats capture messages used to validate the client's startup configuration, LF3 captures the clients end-time, and LF4 captures subscriber client runtime behavior. Likewise, s_1 and s_2 are used to validate QED server and client node startup configuration, whereas s_3, s_4, s_5 and s_6 are used validate runtime behavior of the subscriber and publisher clients.

4.2 Experiment Configuration

Each experiment for evaluating QED's enhancements to the GIG middleware was executed in ISISlab (`www.isislab.vanderbilt.edu`), which is powered by Emulab [24] software. Emulab enables QED testers to configure network topologies and operating systems to produce a realistic target environment for enterprise DRE system integration testing. Each node in ISISlab is an IBM BladeType L20, dual-CPU 2.8 GHz Xeon processor with 1 GB RAM. Each node used in the QED experiments was configured to run the Fedora Core 6 operating system and implemented network bandwidth management, which was done by modifying Linux kernel settings.

Figure 4 shows the experiment setup on ISISlab. Each experiment

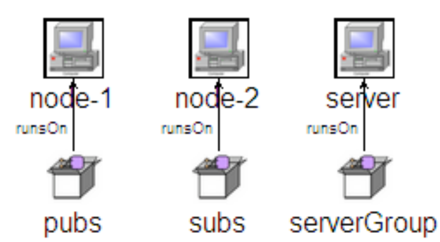

Figure 4: QED Experiment Deployment

was conducted using three ISISlab nodes: one node for running subscriber clients that generated load on the GIG middleware's event dissemination service, one node for running publisher clients that generated load on the GIG middleware's submission service, and one node for running the GIG server. Each experiment ran for 6 minutes using two GIG middleware configurations: (1) the baseline GIG middleware implementation and (2) the enhanced QED/GIG middleware [17] implementation that had adaptive QoS management capabilities integrated into the baseline implementation. We ran the experiments for 6 minutes so their results could be easily compared in terms of throughput (*i.e.*, the total number of information objects received by each client).

Each experiment for validating QED's enhancements to the GIG middleware consisted of (1) three publishing clients that published information objects using the GIG infrastructure and (2) three subscribing clients that received information objects published by the publisher clients. Each publisher client was configured to be either high, medium, or low importance; each subscriber client received information objects from all the publisher clients. Each publisher client has a publication rate of 6 Hz and a payload (*i.e.*, the amount of data transmitted with an event) of 1 KB.

We configured the experiments to create a bandwidth bottleneck by restricting available shared network bandwidth to 320 Kbps for the QED/GIG middleware's information dissemination service. Each QED experiment was replicated using both the baseline implementation (*i.e.*, the GIG middleware without QED enhancements) and the QED/GIG implementation. Our goal was to highlight QED implementation's QoS management capabilities that were not present in the baseline GIG middleware implementation.

QED testers defined validation states to evaluate the experiments test execution correctness, as described in Section 4.1. Likewise, Section 4.2 discussed the different configurations QED testers evaluated with the QED experiments. To evaluate the execution correctness of their tests, however, QED testers had to determine a strategy for performing each evaluation.

The QED testers therefore divided the test execution steps into the following groups: (1) environment initialization on each node (*e.g.*, setting environment variables on each node for the desired GIG configuration and installing desired GIG components, (2) server startup for desired GIG configuration, and (3) publisher and subscriber client start-up with desired publish rate, predicates, and event payloads. Likewise, for experiment validation, test correctness states (see Listing 3) were divided into the following categories:

1. **ENV INIT**, which are environment states used to validate that the proper environment variables are set on each publisher and subscriber node. The example state s_1 in Listing 3 validates that the environment file used for the given experiment sets the environment variables listed for QED/GIG middleware;

2. **PUB INIT**, which are publisher initialization states that validate the number of publishers running on the experiment node and their publish rates. The example state s_2 in the Listing 3 validates that for publisher id `pub1`, publish rate is set to 6 Hz. States similar to s_2 were also defined for `pub2` and `pub3` in the experiment, respectively;

3. **SUB INIT**, which are subscriber initialization states that validate the number of subscribers running on the experiment node and their startup configuration, *e.g.*, the importance of each subscriber and the predicates strings for filtering information objects;

4. **PUB RUNTIME**, which are publisher runtime states for validating that each publisher maintains the given publish rate and does not fail during the experiment. The example state s_3 in Listing 3 validates that each publisher runs for the entire duration of the experiment, *i.e.*, 6 minutes; and

5. **SUB RUNTIME**, which are subscriber runtime states for validating that each subscriber receives the expected minimum number of information objects with complete payload. The states also check that each subscribers runs for the entire duration of the experiment, *i.e.*, 6 minutes.

The example states in Listing 3 validate that the size of received payloads match the size of payloads sent by the publisher clients. The example states also validate that subscriber clients with id `sub1` and `sub3` receive the minimum number of information objects as per their expected QoS levels. Subscriber client `sub1` is a high importance subscriber and therefore should receive the highest QoS, which is measured it terms of the number of information objects received by a subscriber. Subscriber `sub3` is a low importance subscriber and therefore should receive lower QoS. The minimum number of information objects defined for `sub3` in s_6 is an estimated minimum value to ensure that the lower importance subscriber client is running.

Experiment initialization and environment validation states (*i.e.*, s_1 and s_2) were given highest priority because correct configuration is critical for correct runtime behavior of the experiment. Likewise, runtime states (*i.e.*, s_3 and s_4) are also given equivalent high priority because QED testers considered them as equally important as the initialization states when ensuring correct runtime behavior. State s_3 ensures that all publisher clients run for complete duration of the experiment *i.e.*, 6 minutes and state s_4 ensures that no corrupt or incomplete payloads are received by the subscriber clients.

The runtime states (*i.e.*, s_5 and s_6) are given lower priority because they only measure the runtime behavior of the system. Since the QEG/GIG middleware is a priority-based system, it is necessary that QoS requirements of high importance clients and end-users are given higher preference. It is therefore acceptable if the low-importance subscriber clients do not meet the estimated minimum number of information objects, while high-importance subscriber clients receive their estimated minimum number of information objects. As a result, QED testers selected different relative priorities for states s_5 and s_6.

4.3 Experiment Results

Given the scenarios and configurations discusses in Section 4.2, the TE Score helps QED testers identify failures using system execution traces, as well as perform their desired QoS trade-offs. The following four test cases showcase how the TE Score can be used to quickly identify correct and incorrect executions in experiments.

Case 1: Validating execution correctness of performance evaluation experiment. As explained in Section 4.2, QED testers performed experiments that compared the performance of the baseline GIG middleware to the QED/GIG middleware under normal conditions. The experiments measured throughput (*i.e.*, the number of information objects received by each subscriber) for subscribers of low-, medium-, and high-importance subscribers. The experiments also demonstrated that the baseline GIG middleware did not provide differentiated QoS of services to subscribers with varying importance, whereas the QED/GIG middleware provided such services.

Table 1 presents the execution correctness score calculated by the TE Score for one execution of this particular experiment. As shown

Table 1: Results for QoS Performance Evaluation Using the TE Score

	ENV INIT SCORE	PUB INIT SCORE	SUB INIT SCORE	PUB RUN-TIME SCORE	SUB RUN-TIME SCORE
gig	100%	100%	100%	100%	50%
qed-gig	100%	100%	100%	100%	100%

in this table, the initialization states succeed. The QED testers were therefore confident the experiment initialized correctly on all nodes. The table, however, shows that the runtime states for the subscriber (*i.e.*, SUB RUNTIME SCORE) was different for the baseline GIG middleware and GIG/QED middleware. For the baseline GIG middleware, the execution correctness score was 50%, whereas the execution correctness score for the GIG/QED middleware was 100%. Since the baseline GIG middleware does not provide differentiated services to clients with varying importance, all subscribers receive the same number of information objects—resulting in only 50% execution correctness.

When the same experiment was run using QED/GIG middleware, however, 100% execution correctness was observed. The TE score for subscriber runtime states thus highlights the difference between the baseline GIG middleware and QED/GIG middleware capabilities.

Case 2: Validating execution correctness for client configuration. As explained in Section 4.2 and Case 1, the QED testers ran the same experiment using the QED/GIG middleware. When switching the middleware for the experiments, in some cases, one of the three publishers failed to start on the publisher node due to low memory availability from the previous experiment. Due of this failure, few information object were published and received by the publisher and subscribers, respectively. Moreover, the experiment did not execute correctly or to completeness.

Table 2 presents the correctness score calculated by the TE Score for one execution of this particular case. As shown in this table, the

Table 2: Results for client configuration evaluation using the TE Score

	ENV INIT SCORE	PUB INIT SCORE	SUB INIT SCORE	PUB RUN-TIME SCORE	SUB RUN-TIME SCORE
gig	100%	100%	100%	100%	50%
qed-gig	100%	0%	100%	33%	80%

baseline QED middleware experiment executed as expected since it has the same results from Table 1 in Case 1. The execution correctness for the GIG/QED middleware experiment, however, did not score well.

Table 2 also showed how the publishers failed to initialize. Due to this failure, subscribers did not receive the correct number of events—thereby having a low TE Score. Since QED testers were using the TE Score they could quickly learn and troubleshoot the problem was occurring when switching between experiments that compared the baseline GIG middleware to the GIG/QED middleware.

Case 3: Validating execution correctness of environment configuration. After running experiments with baseline GIG middleware, the QED testers had to update the environment configuration on all nodes in the experiment so they could execute the replicated experiment(s) using the enhanced QED/GIG implementation. Due to configuration errors (*e.g.*, errors in the automation script and incorrect parameters passed to the automation script) the environment update did not always succeed on each node, such as the node hosting the subscriber. Due to these errors, subscribers would be run on their target node with invalid configurations. Moreover, the subscribers failed to receive any information objects from the publisher clients.

Table 3 presents the correctness score calculated by the TE Score for one execution of this particular case. As shown in this table,

Table 3: Results for environment configuration evaluation using TE Score.

	ENV INIT SCORE	PUB INIT SCORE	SUB INIT SCORE	PUB RUN-TIME SCORE	SUB RUN-TIME SCORE
gig	100%	100%	100%	100%	50%
qed-gig	0%	100%	100%	100%	0%

the experiments for the baseline GIG middleware executed as expected. The experiments for the GIG/QED middleware, however, did not execute as expected.

Table 3 also shows that the environment initialization states (*i.e.*, ENV INIT SCORE) was 0%, meaning none of its execution states were reached. Likewise, because the environment initialization states failed in this case, the subscriber runtime states (*i.e.*, SUB RUNTIME SCORE) failed to execute. Since the QED testers used the TE Score to validate the execution correctness, they quickly learned that the error was located on the subscriber node in this case.

Case 4: Validating execution correctness when operating in resource constrained environments. As discussed in Section 4.2, the QED testers wanted to compare capabilities of the baseline GIG middleware and the enhanced GIG/QED middleware in resource constrained environments, such as limited bandwidth for sending events (or information objects). The baseline GIG implementation does not support any adaptation capabilities when operating in resource constrained environments. In contrast, the GIG/QED middleware is designed to adapt to such conditions and ensure clients meet their QoS requirements with respect to their level of importance. If the GIG/QED middleware cannot ensure all subscribers will meet their requirements, then the GIG/QED middleware will ignore the QoS requirements of low importance subscribers higher importance subscribers can continue to meet their QoS requirements.

When the experiment was run with QED/GIG implementation, lower importance subscribers received fewer information objects than the minimum number of information objects defined in state s_6 in

Listing 3. Thus s_6 failed for this test during test validation.

Table 4 presents the correctness score calculated by the TE Score for one execution of this particular case. As shown in this table,

Table 4: Results for QoS trade-off analysis using TE Score

	ENV INIT SCORE	PUB INIT SCORE	SUB INIT SCORE	PUB RUN-TIME SCORE	SUB RUN-TIME SCORE
gig	100%	100%	100%	100%	50%
qed-gig	100%	100%	100%	100%	87%

the SUB RUNTIME SCORE is of most importance since the execution states in the other categories was reached. The SUB RUNTIME SCORE is calculated using s_6 in Listing 3. Since the experiment using the GIG/QED middleware did not receive the correct number of events, Table 4 shows only partial success for subscribers runtime validation states. This test, however, is still be considered valid (and correct), since the service to higher importance subscribers still received events under resource constrained conditions. If a higher importance subscriber had crashed at runtime or received fewer events than expected, then SUB RUNTIME SCORE would be much lower because the runtime execution state for higher importance subscribers has a higher priority when compared to runtime execution states for medium and lower importance subscribers.

These test results show how the TE Score simplifies the evaluation and trade-off analysis of QoS performance for experiments of the QED/GIG middleware. Without the TE Score, QED testers would have had to manually identify configuration and runtime failures in the experiments. Moreover, QED testers would have had to manually perform trade-off analysis between the different execution states of the experiments. By leveraging the TE Score to assist with their efforts, QED testers could focus more on defining and running experiments to validate their enhancements to the GIG middleware, as opposed to dealing with low-level testing and evaluation concerns (such as gathering and analyzing data collected from many software components executing on many different hardware node).

5. RELATED WORK

This section compares TE Score with other related work on correctness testing, QoS trade-off analysis, and leveraging system execution traces for validation purposes.

System execution traces. Moe et al. [21] present a technique for using system execution traces to understand distributed system behavior and identify anomalies in behavior. Their technique uses intercepters, which is a form of "black-box" testing, to monitor system events, such as sending/receiving an event. TE Score differs from their technique in that is uses a "white-box" approach to understanding behavior. This is because metrics used to validate test behavior comes from data generated inside the actual component (*i.e.*, the log messages). This offers a richer set of data to perform analysis, understand distributed system behavior, and detect anomalies in the behavior that may not be detectable from "black-box" testing alone.

Chang et al. [4] show how system execution traces can be used to validate software functional properties. For example, their technique uses *parameterized patterns* [2] to data mine system execution traces and validate functional correctness to test execution. TE Score is similar in that is uses system execution traces to capture and extract metrics of interest. TE Score is different in that is focuses on validating correctness of QoS test execution, which involves evaluating QoS execution states of the system.

Correctness testing. Many conventional techniques are used for correctness testing of enterprise DRE systems, such as assertion-based testing [5], continuous integration [8], and unit testing [19]. Irrespective of the correctness testing approach, conventional techniques focus on the functional concerns of enterprise DRE systems. TE Score differs from conventional approaches in that it focuses on ensuring correctness in QoS test execution. In addition, TE Score can also ensure correctness in functional properties as do many existing conventional techniques if the necessary data is captured in system execution traces.

Tian et al. [30] present a reliability measurement for providing reliability assessment for large-scale software systems. Their technique uses failure detections in collected data to not only assess the overall reliability of the system, but also track testing progress in addressing identified defects in the software. TE Score is similar in that it provides a measurement for assessing the correctness (or reliability) of QoS test execution, and identifying where improvements are needed. TE Score, however, differs in that it focuses on assessing QoS test execution, which is based on QoS properties that influence each other and cannot be assessed as disjoint concerns like functional properties.

Trade-off analysis. Lee et al. [15] present an approach for conducting trade-off analysis in requirements engineering for complex systems. Their approach assists developers in measuring how different requirements influence each other. TE Score is similar in that its weighted grading system assist developers in conducting trade-off analysis between different QoS execution states. TE Score differs from Lee's work in that TE Score measures how conflicting concerns affect the entire solution (*i.e.*, correctness of QoS execution test) whereas Lee's work measures how different requirements affect each other.

6. CONCLUDING REMARKS

The ability to quantify the degree of correctness when executing QoS tests helps increase confidence levels in QoS assurance since DRE system testers need not rely on *ad hoc* techniques to ensure correctness properties exist in their QoS tests. This paper presented and evaluated a methodology called the *Test Execution (TE) Score* whose aim is to quantify the correctness of QoS test execution. DRE system testers use the TE Score to define correctness tests that take into account the different QoS execution states of the system. Correctness tests also take into consideration that different QoS execution states have different priorities. DRE system testers therefore can perform trade-off analysis within their correctness tests to ensure that more important QoS execution states have greater influence on the results.

Based on our results and experience developing and applying TE Score to a representative enterprise DRE system, we learned the following lessons:

- **Manually specifying the execution states helped reduce false negatives** because TE Score was not trying to deduce them automatically. More importantly, it helped DRE system testers understand the test execution process better by identifying *important* states that should influence overall correctness in QoS test execution.

- **Time-based correctness of QoS execution testing is needed** because QoS properties can change over time. In some cases, DRE system testers many want to ensure correctness of QoS test execution at different time slices using different QoS execution states. Our future work will therefore investigate techniques for leverage temporal-logic [13] to facilitate time-based correctness testing of QoS test execution.

- **Execution state-based specification helped perform trade-off analysis** it allowed finer control over how different states affect the final analysis. More importantly, assigning priorities to the different execution states helped improve DRE system testers control how much affect a given state had on the final analysis.

The TE Score, CUTS, and UNITE are freely available in open-source format for download from `www.cs.iupui.edu/CUTS`.

7. REFERENCES

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

[2] B. S. Baker. Parameterized Pattern Matching by Boyer-Moore-type Algorithms. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 541–550, 1995.

[3] F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[4] F. Chang and J. Ren. Validating system properties exhibited in execution traces. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 517–520, New York, NY, USA, 2007. ACM.

[5] Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 231–255, London, UK, 2002. Springer-Verlag.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[7] E. Downs, P. Clare, and I. Coe. *Structured Systems Analysis and Design Method: Application and Context*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.

[8] M. Fowler. Continuous Integration. www.martinfowler.com/articles/ continuousIntegration.html, May 2006.

[9] J. H. Hill, J. Slaby, S. Baker, and D. C. Schmidt. Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[10] 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.

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

[12] Institute for Software Integrated Systems. Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO/, Vanderbilt University.

[13] L. Lamport. The Temporal Logic of Actions. *ACM Transactions of Programming Languages and Systems*, 16(3):872–923, 1994.

[14] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[15] J. Lee and J.-Y. Kuo. New Approach to Requirements Trade-Off Analysis for Complex Systems. *IEEE Transactions on Knowledge and Data Engineering*, 10(4):551–562, 1998.

[16] N. X. Liu and J. S. Baras. Modelling multi-dimensional qos: Some fundamental constraints: Research articles. *International Journal of Communication Systems*, 17(3):193–215, 2004.

[17] J. P. Loyall, M. Gillen, A. Paulos, L. Bunch, M. Carvalho, J. Edmondson, P. Varshneya, D. C. Schmidt, and A. Martignoni. Dynamic Policy-Driven Quality of Service in Service-Oriented Systems. In *Proceedings of the 13th International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC '10*, Carmona, Spain, May 2010.

[18] J. Mann. *The Role of Project Escalation in Explaining Runaway Information Systems Development Projects: A Field Study*. PhD thesis, Georgia State University, Atlanta, GA, 1996.

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

[20] 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.

[21] J. Moe and D. A. Carr. Understanding Distributed Systems via Execution Trace Data. In *International Workshop on Program Comprehension*, 2001.

[22] Prism Technologies. OpenSplice Data Distribution Service. www.prismtechnologies.com/, 2006.

[23] Real-time Innovations. NDDS: The Real-time Publish-Subscribe Middleware. www.rti.com/products/ndds/ndwp0899.pdf, 1999.

[24] R. Ricci, C. Alfred, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. *SIGCOMM Computer Communications Review*, 33(2):30–44, Apr. 2003.

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

[26] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc., New York, NY, USA, 1994.

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

[28] A. Snow and M. Keil. The Challenges of Accurate Project Status Reporting. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, 2001.

[29] I. M. Technical and I. Majzik. Software Monitoring and Debugging Using Compressed Signature Sequences. In *Proceedings of the* 22^{nd} *EUROMICRO Conference*, pages 311–318, September 1996.

[30] J. Tian, P. Lu, and J. Palma. Test-execution-based reliability measurement and modeling for large commercial software. *IEEE Transactions on Software Engineering*, 21(5):405–414, 1995.