Central European Journal of Computer Science

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

Research Article

James H. Hill[1]*, Pooja Varshneya[2]†, Douglas C. Schmidt[2]‡

1 Indiana University-Purdue University Indianapolis,
Indianapolis, IN 46202, USA

2 Vanderbilt University,
Nashville, TN, USA

Abstract: Effective validation of distributed real-time and embedded (DRE) system quality-of-service (QoS) properties (*e.g.*, event prioritization, latency, and throughput) requires testing system capabilities in representative execution environments. Unfortunately, evaluating the correctness of such tests is hard since it requires validating many states dispersed across many hardware and software components. To address this problem, this article presents a method called *Test Execution (TE) Score* for validating execution correctness of DRE system tests and empirically evaluates TE Score in the context of a representative DRE system. Results from this evaluation show that TE Score can determine the percentage correctness in test execution, and facilitate trade-off analysis of execution states, thereby increasing confidence in QoS assurance and improving test quality.

Keywords: distributed real-time and embedded (DRE) systems, test correctness, trade-off analysis, system execution traces

1. Introduction

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) [1]. It is therefore becoming more critical to validate their 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 resolve performance bottlenecks with less time and effort than deferring such validation until final system integration [2, 3].

* *E-mail: hillj@cs.iupui.edu*
† *E-mail: p.varshneya@vanderbilt.edu*
‡ *E-mail: d.schmidt@vanderbilt.edu*

System execution modeling (SEM) [4] is a promising approach for continuously validating DRE system QoS properties throughout their software lifecycles. SEM tools enable DRE system developers to (1) model system behavior and workload at high-levels of abstraction and (2) use these models to validate QoS properties on the target architecture. Although SEM tools can provide early insight a DRE system's QoS values, 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 [5], *e.g.*, failure to detect the effects of node failures on QoS properties because injected faults did not occur as expected. Likewise, DRE systems have many competing and conflicting QoS properties that must be validated [1]. 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 (such as event timestamps, application state, and network interface stats) 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 [6]. 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 DRE system QoS properties in their target environments.

Solution approach → Correctness validation via system execution traces. System execution traces [7] are artifacts of executing a software system (*e.g.*, a DRE system) in a representative 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 what component exceeded its allotted execution time. In the context of correctness validation, system execution traces can help quantify the correctness of a DRE system test that validates QoS properties in terms of its states and trade-off analysis of such properties.

This article therefore describes a method called *Test Execution (TE) Score* that uses system execution traces to validate DRE system test correctness. DRE system testers use 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. TE Score then uses system execution traces to evaluate test correctness using the specified (in)valid state and QoS properties. Results from our experiments show how applying TE Score to a representative DRE system can provide DRE system testers with a correctness grade (*i.e.*, a percentage) that quantifies how well their tests execute. Moreover, TE Score helps identify test errors that must be resolved to improve correctness and increase confidence in QoS assurance for DRE systems.

Article organization. The remainder of this article is organized as follows: Section 2 introduces a representative DRE system case study to motivate the need for TE Score; Section 3 describes the structure and functionality

of TE Score; Section 4 analyzes the results of experiments that applied TE Score to the case study; Section 5 compares TE Score with related work; and Section 6 presents concluding remarks.

2. Case Study: The QED Project

The *QoS-Enabled Dissemination* (QED) [8] project is a multi-year, multi-team effort to create and evaluate information management middleware to meet the QoS requirements of component-based DRE systems in the Global Information Grid (GIG) [9]. The GIG is a large-scale DRE system [1] designed to ensure that different applications collaborate effectively and deliver appropriate information to users in a timely, dependable, and secure manner. Figure 1 shows QED in context of the GIG.

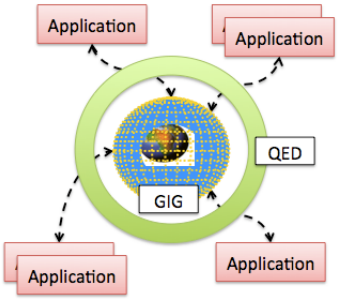

Figure 1. QED Relationship to the GIG

One of the key challenges faced by the QED development team was overcoming the *serialized-phasing development problem* [10], 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 flaws [2, 3]. To overcome this problem, QED testers used the CUTS and UNITE (see Sidebar 1) SEM tools to validate QoS properties of the QED middleware on existing GIG middleware continuously throughout the software lifecycle [11].

Although CUTS and UNITE enabled QED testers to evaluate QED's enhancements relative to existing GIG middleware, CUTS was bound to the same limitations of evaluating QoS test correctness discussed in Section 1. In particular, QED testers were faced with the following challenges:[1]

Challenge 1: Inability to validate execution correctness of QoS test. Executing a QoS test on a DRE system 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

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

Sidebar 1: Overview of CUTS and UNITE

The *Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS)* [12] is a system execution modeling tool for large-scale DRE systems. It enables DRE system testers to validate QoS properties on the target architecture during early phases of the software lifecycle. DRE system testers use CUTS as follows:

1. Use domain-specific modeling languages [13] to model behavior and workload at high-levels of abstraction;
2. Use generative programming techniques [14] 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 in its target execution environment.

DRE system testers can also replace emulated portions of the system with its real counterpart as its development is completed through a process called *continuous system integration testing*.

The *Understanding Non-functional Intentions via Testing and Experimentation* (UNITE) [11] tool is distributed with CUTS that mines distributed system execution traces to validate QoS properties. UNITE also constructs QoS performance graphs that illustrate data trends throughout the lifetime of the system (*i.e.*, how a QoS property changed with respect to time). Section 3.1 provides a more detailed overview of UNITE.

correctly.

QED testers therefore needed a method that simplified validating whether QoS validation tests execute correctly. Ideally, this method would 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). Section 3.2 describes how TE Score method addresses this challenge problem using state-based specifications.

Challenge 2: Inability to perform trade-off analysis between QoS properties. QoS properties are a multi-dimension concern [15]. It is hard to simultaneously ensure all 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 existing GIG middleware.

After QED testers validated the correctness of QoS test execution (*i.e.*, they address challenge 1), they wanted to validate multiple QoS properties simultaneously because it is time-consuming to validate a single QoS property in isolation. QED testers therefore needed a method to assist in this trade-off analysis between multiple dimensions of QoS properties. Moreover, the method 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 3.3 describes how TE Score addresses this challenge by adding priorities and weights to the state-based specifications.

3. The Structure and Functionality of TE Score

This section describes the structure and functionality of TE Score. Examples from the QED case study introduced in Section 2 are used throughout this section to showcase the applicability of TE Score to DRE systems.

3.1. Mining System Execution Traces with Dataflow Models

Before discussing the details of TE Score, it is first helpful to understand how dataflow models can be used to mine system execution traces and validate QoS properties. As mentioned in Section 1, system execution traces are a collection of messages that capture a DRE system's behavior and state at any given point in time while executing in its target environment. Prior research efforts [11, 16] yielded a SEM tool called UNITE (see Sidebar 1) that mines system execution traces to (1) validate QoS properties and (2) generate QoS performance graphs for QoS properties by viewing its data trend (*i.e.*, how a metric changes with respect to time).

To validate a QoS property and view its data trend using system execution traces, UNITE uses *dataflow models* [17]. A dataflow model captures how data is passed (or flows) through a given domain—similar to how connections between separate components in a DRE system capture how data flows through a distributed system. UNITE defines this dataflow model as $DM = (LF, CR)$ as:

- A set LF of log formats that have a set V of variables identifying what data to extract from messages in a system execution traces. These log formats will identify many occurrences of similar messages in a system execution trace where the 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 [18]. These relations help determine how data flows across different application domains, such as one component sending an event to another component deployed on a different host.

Figure 2 shows a user-defined dataflow model that can produce QoS performance graphs from a system execution trace, such as the one shown in Table 1. This figure shows how two log formats will locate messages for event publication (*i.e.*, LF1 will identify message 45, 46, and 48 in Table 1) and event receipt (*i.e.*, LF2 will identify message 47, 49, and 50 in Table 1). The variables in the two log formats capture the variable portion of each log message, which also contain metrics of interest.

```
AVG(LF2.recvTime - LF1.sendTime)
```

Listing 1. **Expression for calculating latency using UNITE.**

Listing 1 presents the expression specified by DRE system testers to calculate the latency for sending an event based on the data captured by the system execution trace shown in Table 1. UNITE uses the dataflow model and system execution trace to evaluate the expression and optionally view its data trend by removing the aggregation function (*i.e.*, `AVG`).

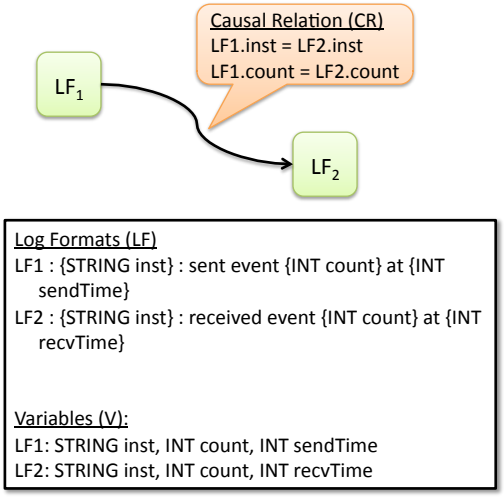

Figure 2. **Example dataflow model in UNITE.**

Table 1. **Data table of an example system execution trace to be analyzed by UNITE.**

ID	Time of Day	Hostname	Severity	Message
45	2009-03-06 05:15:55	node1.isislab.vanderbilt.edu	INFO	Config: sent event at 5 at 120394455
46	2009-03-06 05:15:55	node1.isislab.vanderbilt.edu	INFO	Planner: sent event at 6 at 120394465
47	2009-03-06 05:15:55	node2.isislab.vanderbilt.edu	INFO	Planner: received event at 5 at 120394476
48	2009-03-06 05:15:55	node1.isislab.vanderbilt.edu	INFO	Config: sent event at 7 at 120394480
49	2009-03-06 05:15:55	node2.isislab.vanderbilt.edu	INFO	Effector: received event at 6 at 120394488
50	2009-03-06 05:15:55	node2.isislab.vanderbilt.edu	INFO	Planner: received event at 7 at 120394502

3.2. Specifying QoS Test Execution States

Validating QoS test correctness requires evaluating a 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 evaluation is necessary because QoS properties, such as latency and end-to-end response time, often fluctuate over time due to system dynamics and hardware/software contention. There can also be states that the DRE system must reach (and maintain) to evaluate QoS properties properly, *e.g.*, a component receiving the correct number of events within a time period to evaluate end-to-end response time is under expected workload conditions.

The execution state of a DRE system is the value of its variables at any given point in time. Since it can be hard to capture the value of each variable in a DRE system at a given point in time (*i.e.*, take a global snapshot [18]), system execution traces may be needed to capture the necessary information. Based on the discussion in Section 3.1, it is also possible to extract the needed information and validate test execution correctness. We therefore formally define an execution state of a DRE system as $s(DM) = (V', P)$:

- A set of variables V' for evaluating the system's state of interest where $V' \in V$ in the dataflow model DM; and

- A proposition P that captures the context and expected value of 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 and E_v defines the actual value (or effect) for the given context. It is necessary for the execution state to define a proposition because the same effect can occur in many different contexts. The proposition therefore prevents incorrect data from being extracted from the totality of the data available in a system execution trace.

Implementing QoS test execution states in TE Score. To realize test execution states in TE Score, we leverage UNITE's capabilities for specifying dataflow models. In particular, using the set of variables defined in UNITE's dataflow model, QED testers select variables to define execution states that can validate the execution correctness of their QoS test. This execution state is specified as a proposition P where the context C_v is an expression that defines the scope of the evaluation and E_v is the expected value for the specified context.

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

Listing 2. **Example State Specification for TE Score**

Listing 2 shows an example that validates if 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. They then define a proposition where the context checks the value captured in the variable named `LF1.instName`. If the specified context is valid, *i.e.*, C_v is true, the expected value of the context is evaluated, *i.e.*, E_v is tested. Likewise, the execution state is ignored if the context is invalid.

3.3. Specifying Execution Correctness Tests

Section 3.2 discussed how TE Score defines a single execution state for validating execution correctness. 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 workloads. Likewise, it may be necessary to check that test execution does not reach an invalid state, but it may be easier to express this requirement in terms of invalid states, as opposed to valid states.

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

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

- s is the target QoS test execution state;
- 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 to assigning priorities;
- min is the minimum number of occurrences the specified state s can occur throughout the execution of the QoS test (*i.e.*, the state's lower bound); and
- max is the maximum number of occurrences that the specified state s can occur throughout the execution of the QoS test (*i.e.*, the state's upper bound).

If the state type is *invalid* min and max play a different role, where the specified range $[min, max]$ represents the number of occurrences that cause the execution state to fail. Finally, an execution correctness test CT is defined as a set S of QoS validation states where $s' \in S$.

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 execution correctness test is given a priority that determines its level of importance when conducting trade-off analysis between different execution states.

s'_1 :

s: $(LF3.instName = Receiver) \rightarrow LF4.eventCount/(LF5.stopTime - LF6.startTime) > 5$

t: `valid`

p: 2

min: 1

max: 1

s'_2 :

s: $(LF1.instName = ConfigOp) \rightarrow (LF2.sendTime - LF1.recvTime) > 30$

t: `invalid`

p: 4

min: 1

max: unbounded

s'_3 :

s: $LF7.endTime - LF8.beginTime < 100$

t: `valid`

p: 1

min: 0

max: unbounded

Listing 3. **Example Correctness Test Specification in the TE Score**

Listing 3 shows an example of a correctness test that validates the correctness of QoS test execution. This example contains the following different QoS execution states:

1. A state validating the arrival rate of events into a component named `Receiver` is 5 Hz, which should only occur once;
2. A state validating that response time for a component named `ConfigOp` is never greater than 30 msec; 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.

3.4. Evaluating Execution Correctness Test Specifications

After defining an execution correctness test specification (see Section 3.3), the final step in the evaluation process is evaluating it. The main goal of this 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 two purposes: (1) it helps DRE system testers determine how well tests meet their expectations and (2) it identifies areas where tests may need improvement.
Given Section 3.3's definition of an execution correctness test specification, 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, 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, overall evaluation should be impacted negatively 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. Equation 1 calculates the weight of a given state s' in an execution correctness test 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 execution correctness test grade and reaching invalid states decreases the grade. Likewise, the grade is decreased if a valid state is not reached, whereas the resulting grade is increased if an invalid state is not reached.

Equations 2 and 3 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.

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

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

Finally, Equation 4 and Equation 5 determine the final grade G for an execution correctness test specification.

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

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

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

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

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

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

Algorithm 1 TE Score's 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
```

To prevent false negatives from occurring in an evaluation, 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, if the bounds of occurrence for s' are [0, unbounded] that state should always occur. When evaluating the negation of the expected value E_v the result set should be empty.

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

4. Applying TE Score to the QED Project

This section analyzes the results of applying 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, QED aims to enhance QoS concerns of GIG middleware by adding adaptive information management capabilities to it. To ensure the QED project does in fact improve the QoS capabilities of GIG middleware, QED testers constructed several experiments to evaluate QED's enhancements. In particular, QED

testers wanted to evaluate the following QED capabilities:

- **Prioritized services for high priority users.** Experiments were designed to verify higher importance subscribers received prioritized services when compared to lower importance subscribers.
- **Adaptive resource management capabilities to ensure subscriber QoS requirements.** Experiments were designed to simulate resource constrained environments, *e.g.*, limited dissemination bandwidth, to validate the adaptive capabilities of QED middleware and ensure subscriber QoS properties were not degraded when compared to lower importance subscribers.
- **QoS performance measurement for higher importance versus lower importance subscribers.** Experiments were designed 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 and analyze, respectively, several experiments that evaluated these concerns empirically. The experiments contained up to 40 software components running on hardware components communicating via a shared network. The application components for the experiments were first modeled using CUTS's behavior and workload modeling languages [19]. As shown in Figure 3, each software component has behavior that would exercise some aspect of the QED/GIG middleware, *e.g.*, sending high payloads, changing application event prioritization at runtime, occupying CPU time to influence resource contention.

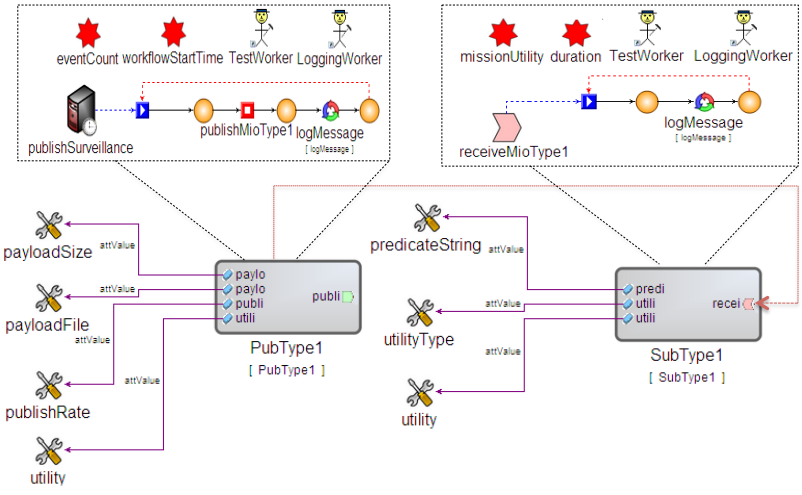

Figure 3. **QED Clients Modeled Using CUTS**

Finally, QED testers used TE Score to evaluate different execution states of QED experiments (see Section 3.2 and Section 3.3) and grade the quality of each experiment based on its execution states (see Section 3.4). Listing 4 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 4. **Examples of the TE Score States**

As shown in Listing 4, LF1,..., LF4 define log messages that capture client 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 that 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 validate QED server and client node startup configuration, whereas s_3, s_4, s_5 and s_6 validate runtime behavior of the subscriber and publisher clients. In total, over 10 different log formats and validation states were used in the experiments.

4.2. Experiment Configuration

Each experiment was executed in ISISlab (www.isislab.vanderbilt.edu), which contains over 100 computing cores powered by Emulab (www.emulab.net) software. Emulab enables QED testers to configure network topologies and operating systems to produce a realistic target environment for 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 Fedora Core 6 and implemented network bandwidth management by modifying Linux kernel settings.

Each experiment used the baseline GIG middleware implementation and the enhanced QED/GIG middleware [8] implementation that added adaptive QoS management capabilities to the baseline. Each experiment also consisted of (1) multiple clients that published information objects using the GIG infrastructure and (2) multiple subscribing clients that received information objects published by the publisher clients. Each publisher client was configured as 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 at most 6 Hz and a payload (*i.e.*, the amount of data transmitted with an event) of at most 1 KB.

We ran the experiments for 6 minutes so their results could be compared in terms of throughput (*i.e.*, the total number of information objects received by each client). Our goal therefore was to highlight QED implementation's QoS management capabilities that were absent in the baseline GIG middleware implementation. To evaluate execution correctness, QED testers divided their test execution steps into the following groups: initialization and runtime. This decomposition provided finer grain evaluation for evaluating their test execution correctness. The initialization group consisted of the following categories:

- **ENV INIT (EI)**, which are environment states that validate environment variables are set on each publisher and subscriber node. The example state s_1 in Listing 4 validates that the environment file used for

the given experiment sets the environment variables listed for QED/GIG middleware;

- **PUB INIT (PI)**, 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 4 validates that publisher `pub1` has a publish rate of 6 Hz. States similar to s_2 were also defined for `pub2` and `pub3` in the experiment; and

- **SUB INIT (SI)**, 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.

Likewise, the runtime group consisted of the following categories:

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

- **SUB RUNTIME (SR)**, which are subscriber runtime states that validate 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 4 validate that the size of received payloads match the size of payloads sent by 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) receive the highest priority because correct configuration is critical for correct runtime behavior of the experiment. Likewise, runtime states (*i.e.*, s_3 and s_4) were 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 runtime behavior of the system. Since the QED/GIG middleware is priority-based, QoS requirements of high importance clients 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

The following four test cases showcase how QED testers used TE Score to 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/GIGbmiddleware provided such differentiated QoS of services.

Table 2. **Results for QoS Performance Evaluation Using TE Score**

	EI	PI	SI	PR	SR
gig	100%	100%	100%	100%	50%
qed-gig	100%	100%	100%	100%	100%

Table 2 presents the execution correctness score calculated by TE Score for one execution of this particular experiment. As shown in this table, the initialization states succeed. 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 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, QED testers ran the same experiments using QED/GIG middleware. When switching the middleware publishers failed to start in some cases due to low memory availability from the previous experiment. Due to this failure, few information objects were published and received by publishers and subscribers, respectively. Moreover, the experiment neither executed correctly nor to completion.

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

in this table, the baseline QED middleware experiment executed as expected since it has the same results from Table 2 in Case 1. The execution correctness for the GIG/QED middleware experiment, however, did not score well.

Table 3. Results for Client Configuration Evaluation Using TE Score

	EI	PI	SI	PR	SR
gig	100%	100%	100%	100%	50%
qed-gig	100%	0%	100%	33%	80%

Table 3 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 score. Since QED testers used TE Score, they quickly pinpointed that the problem was occurring when switching between experiments.

Case 3: Validating execution correctness of environment configuration. After running experiments with baseline GIG middleware, 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. As a result of these errors, subscribers would run on their target node with invalid configurations. Moreover, the subscribers failed to receive any information objects from publishers.

Table 4 presents the correctness score calculated by TE Score for one execution of this particular case. As shown in this table, the experiments for the baseline GIG middleware executed as expected. The experiments for the GIG/QED middleware, however, did not execute as expected.

Table 4. Results for Environment Configuration Evaluation Using TE Score

	EI	PI	SI	PR	SR
gig	100%	100%	100%	100%	50%
qed-gig	0%	100%	100%	100%	0%

Table 4 also shows that the environment initialization states was 0%, meaning none of its execution states were reached. Likewise, because the environment initialization states failed in this case, the subscriber runtime states failed. Since the QED testers used TE Score to validate execution correctness, they quickly pinpointed 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, 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 it will ignore QoS requirements of low importance subscribers so higher importance subscribers can continue meeting their QoS requirements. When experiments were run with QED/GIG implementation, lower importance subscribers received fewer information objects than the minimum number of information objects defined for state s_6 in Listing 4. Thus, s_6 should fail for this test.

Table 5. Results for Trade-off Analysis of Execution States Using TE Score

	EI	PI	SI	PR	SR
gig	100%	100%	100%	100%	50%
qed-gig	100%	100%	100%	100%	87%

Table 5 presents the correctness score calculated by TE Score for one execution of this particular case. As shown in this table, SUB RUNTIME SCORE is of most importance since execution states in the other categories were reached. Since the experiment using GIG/QED middleware did not receive the correct number of events, Table 5 shows only partial success for subscriber 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. This result occurs 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.

Synopsis. These test results show how TE Score simplified the evaluation and trade-off analysis of QoS performance for experiments of the QED/GIG middleware. Without TE Score, QED testers would have to identify configuration and runtime failures manually in the experiments. Moreover, QED testers would have to perform trade-off analysis manually between the different execution states of the experiments. By leveraging 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.

5. Related Work

This section compares our work on TS Score with related work on system execution traces, conventional correctness testing, and trade-off analysis.

System execution traces. Moe et al. [20] 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 this technique since it uses a "white-box" approach to understand behavior, where metrics are used to validate test behavior comes from data generated inside the actual component (*i.e.*, the log messages). TE Score's approach

offers a richer set of data to perform analysis, understand DRE system behavior, and detect anomalies in the behavior that may not be detectable from black-box testing alone.

Chang et al. [7] show how system execution traces can be used to validate software functional properties. For example, their technique uses *parameterized patterns* [21] to 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. It differs, however, by focusing on validating correctness of QoS test execution, which involves evaluating execution states of the system.

Conventional correctness testing. Many conventional techniques, such as assertion-based testing [6], continuous integration [22], and unit testing [23], have been used to test the correctness of DRE systems. Irrespective of the correctness testing approach, conventional techniques focus on non-distributed systems. TE Score differs from conventional approaches by focusing on DRE systems. This focus incurs additional complexity since data must be correlated correctly across the entire system. TE Score can also behave like conventional correctness testing approaches if the necessary data is captured in system execution traces.

Tian et al. [24] 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. It differs, however, 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. [25] 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. It differs from Lee et al.'s work, however, since TE Score measures how conflicting concerns affect the entire solution (*i.e.*, correctness of QoS execution test) whereas Lee et al.'s work measures how different requirements affect each other.

6. Concluding Remarks

The ability to quantify the degree of correctness for QoS tests helps increase confidence levels in QoS assurance. This article presented a method called *Test Execution (TE) Score* that quantifies the correctness of QoS test execution. We showed how DRE system testers in the QED project used TE Score to define correctness tests that account for the different execution states of the system and consider that different execution states have different priorities. QED testers used TE Score to perform trade-off analysis within their correctness tests to ensure that important execution states have greater influence on the results.

Based on our experience applying TE Score to a representative 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.
- **Time-based correctness of 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 execution states. Our future work will therefore investigate techniques that leverage temporal-logic [26] to facilitate time-based correctness testing of QoS test execution.
- **Execution state-based specification helped perform trade-off analysis** because 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.
- **Determining the correct number of execution states is not trivial** because it is unknown ahead of time how many execution states are needed to produce meaningful results. For example, too few execution states can make it hard to show how one execution state influences another, or impacts the over score. Likewise, too many execution states can accidently dilute the impact of other execution states. DRE system testers who use TE Score currently determine the correct number of execution states for a correctness test by starting small and increasing the states and priority as needed (*i.e.*, by trial-and-error). Future research therefore will investigate techniques for deriving the appropriate number of execution states for a correctness test.

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

References

[1] S. E. Institute, Tech. Rep., Carnegie Mellon University, Pittsburgh, PA, USA (2006).

[2] A. Snow and M. Keil, in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences* (Maui, Hawaii, 2001).

[3] J. Mann, Ph.D. thesis, Georgia State University, Atlanta, GA (1996).

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

[5] I. M. Technical and I. Majzik, in *Proceedings of the 22nd EUROMICRO Conference* (1996), pp. 311–318.

[6] Y. Cheon and G. T. Leavens, in *Proceedings of the 16th European Conference on Object-Oriented Programming* (Springer-Verlag, London, UK, 2002), pp. 231–255.

[7] F. Chang and J. Ren, in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (ACM, New York, NY, USA, 2007), pp. 517–520, ISBN 978-1-59593-882-4.

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

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

[10] Rittel, H. and Webber, M., Policy Sciences pp. 155–169 (1973).

[11] J. H. Hill, H. A. Turner, J. R. Edmondson, and D. C. Schmidt, in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation* (Denver, Colorado, 2009).

[12] J. H. Hill, J. Slaby, S. Baker, and D. C. Schmidt, in *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications* (Sydney, Australia, 2006).

[13] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, Computer **34**, 44 (2001), ISSN 0018-9162.

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

[15] N. X. Liu and J. S. Baras, International Journal of Communication Systems **17**, 193 (2004).

[16] J. H. Hill, D. C. Schmidt, J. Edmondson, and A. Gokhale, IEEE Software (2010), ISSN 0001-0782.

[17] 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), ISBN 0-13-854324-0.

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

[19] J. H. Hill and A. S. Gokhale, JSW **2**, 9 (2007).

[20] J. Moe and D. A. Carr, in *International Workshop on Program Comprehension* (2001).

[21] B. S. Baker, in *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms* (1995), pp. 541–550.

[22] M. Fowler, *Continuous Integration*, www.martinfowler.com/articles/ continuousIntegration.html (2006).

[23] V. Massol and T. Husted, *JUnit in Action* (Manning Publications Co., Greenwich, CT, USA, 2003), ISBN 1930110995.

[24] J. Tian, P. Lu, and J. Palma, IEEE Transactions on Software Engineering **21**, 405 (1995).

[25] J. Lee and J.-Y. Kuo, IEEE Transactions on Knowledge and Data Engineering **10**, 551 (1998).

[26] L. Lamport, ACM Transactions of Programming Languages and Systems **16**, 872 (1994).