Unit Testing Non-functional Concerns of Component-based Distributed Systems

James H. Hill, Hamilton A. Turner, James R. Edmondson, and Douglas C. Schmidt
Vanderbilt University
Nashville, TN USA
{j.hill, hamilton.a.turner, james.r.edmondson, d.schmidt}@vanderbilt.edu

Abstract

Unit testing component-based distributed systems traditionally involves testing functional concerns of the application logic throughout the software lifecycle. In contrast, testing non-functional distributed system concerns (e.g., end-to-end response time, security, and reliability) typically does not occur until system integration because it requires a complete system to perform such tests, as well as sophisticated techniques to identify and analyze performance metrics that constitute non-functional concerns. Unit testing non-functional concerns is even harder in an agile development environment, due to the disconnect between high-level system specification and low-level performance metrics.

This paper describes a methodology and tool called Understanding Non-functional Intentions via Testing and Experimentation (UNITE). UNITE is designed to unit test non-functional concerns of three component-based distributed systems. The results from applying UNITE to a component-based distributed system show how it simplifies unit testing and evaluation of non-functional properties during the early stages of the software lifecycle.

1 Introduction

Challenges of distriuted system testing. Unit testing [11, 16] is a software validation technique that involves testing functional properties of software, such as setter/-getter methods of a class. For component-based distributed systems, unit testing traditionally tests the application logic. Testing non-functionally concerns (such as end-to-end response time) of component-based distributed systems typically occurs during the software integration phase since testing such concerns requires a complete system [5]. The separation of testing functional and non-functional concerns in a distributed system creates a disconnect between the two, which can result in unforeseen consequences on project development, such as missed project deadlines due to failure to meet non-functional requirements, as shown in Figure 1.

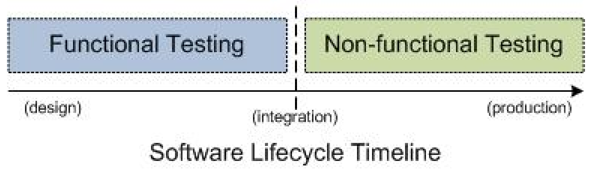

Figure 1. Separation of Functional and Non-functional Testing in Component-based Distributed Systems

System execution modeling (SEM) [3, 6, 10] tools help bridge the gap between understanding how functional and non-functional requirements affect each other. SEM tools (1) model the behavior of a system under development and characterize its workload and (2) provide testers with artifacts to emulate the constructed models on the target architecture and analyze different non-functional concerns, such as end-to-end response time. These tools enable system developers to evaluate non-functional concerns prior to system integration (*i.e.*, before the system is completely developed).

Conventional SEM tools, however, do not provide developers with techniques for unit testing non-functional requirements of a component-based distributed system. Key non-functional requirements include (1) identifying metrics of interest for data collection, (2) extracting such metrics for analysis, and (3) formulating equations based on extracted metrics that analyze an individual non-functional concern or unit test. Moreover, in a development environment where requirements change constantly, system testers must also measure, evaluate, and reason about non-functional concerns at the same level of abstraction as system requirements, and in a similar manner as functional concerns. Unfortunately, conventional SEM tools do not provide testers with techniques for abstracting and mapping low-level metrics to high-level system requirements and specifications. New techniques are therefore needed to unit test non-functional concerns and reasoning about low-level system metrics at a higher-level of abstraction.

Solution Approach → High-level specification and analysis of non-functional concerns. Evaluating non-functional concerns of a component-based distributed system typically requires gathering metrics that are generated at different times while the system is executing in its target environment. For example, understanding a simple metric like system throughput requires capturing data that represents the number of events/users processed, the lifetime of the system, aggregating individual results, and calculating system throughput. For more complex metrics, the evaluation process becomes harder, particularly for large-scale systems or ultra-large-scale systems [12] composed of many components and deployed across many hosts.

This paper describes a methodology and tool called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)* that is designed to alleviate the complexity of specifying and analyzing non-functional concerns of component-based distributed systems. UNITE is based on relational database theory [2] where metrics used to evaluate a non-functional unit test are associated with each other via relations to construct a data table. The constructed metric's table is evaluated by applying an SQL expression based on a user-defined function. Testers can use UNITE to evaluate non-functional properties of their applications via the following steps:

1. Use log messages to capture metrics of interests, such as time an event was sent or values of elements in an event;
2. Identity metrics of interest within the log messages using message constructs, such as: `{STRING ident} sent message {INT eventId} at {INT time}`;
3. Define unit tests to analyze non-functional concerns (such as overall latency, end-to-end response time, or system reliability and security) by formulating equations using the identified metrics, which can span many log messages.

Our experience applying UNITE to a representative component-based distributed system shows it is an effective technique for unit testing non-functional concerns during the early stages of system development. Moreover, as new concerns arise testers need only add new log message(s) to capture the metric(s) along with high-level construct(s) to extract the metric(s). UNITE thus significantly reduces the complexity of specifying non-functional unit tests while producing a repository of historical data that can be analyzed and monitored throughout a distributed system's software development lifecycle.

Paper Organization. The remainder of this paper is organized as follows: Section 2 summarizes a representative distributed system case study used to motivate the need for UNITE; Section 3 describes the structure and functionality of UNITE; Section 4 shows how we applied UNITE to our case study; Section 5 compares UNITE with related work; and Section 6 presents concluding remarks.

2 Case Study: the QED Project

The Global Information Grid (GIG) middleware [1] is a component-based ultra-large-scale system [12]. The GIG is designed to ensure that different applications can collaborate effectively and deliver appropriate information to users in a timely, dependable, and secure manner. Due to the scale and complexity of the GIG, however, conventional implementations do not provide adequate end-to-end quality-of-service (QoS) assurance to applications that must respond rapidly to priority shifts and unfolding situations.

The QoS-Enabled Dissemination (QED) [14] project is a multi-organization collaboration designed to improve GIG middleware so it can meet QoS requirements of users and component-based distributed systems. QED's aims to provide reliable and real-time communication middleware that is resilient to the dynamically changing conditions of GIG environments. Figure 2 shows QED in the context of the GIG. At the heart of the QED middleware is a Java-based information broker based on the Java Messaging Service and JBoss that enables tailoring and prioritizing of information based on mission needs and importance, and responds rapidly to priority shifts and unfolding situations. Moreover, QED leverages technologies such as Mockets [21] and differentiated service queues [7] to provide QoS assurance to GIG applications.

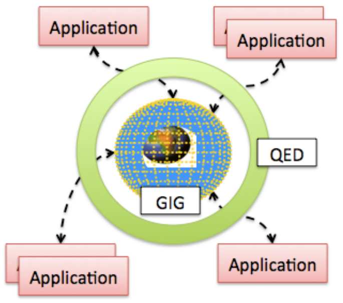

Figure 2. Conceptual Model of QED in the Context of the GIG

The QED project is in its first 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. It is therefore hard for QED developers to ensure their software architecture and implementations are actually

]

improving the QoS of applications that will ultimately run on the GIG middleware. The QED project thus faces a typical problem in large-scale distributed system development: the *serialized phasing problem* [10]. In serialized phasing, the system is developed in layers where components in the upper layer(s) are not developed until long after the components in the lower layer(s) are developed. Design flaws that affect non-functional concerns, however, are typically not discovered until the final stages of development, *e.g.*, at system integration time.

To overcome the serialized-phasing problem, QED developers are using SEM tools to automatically and continuously execute performance regression tests against the QED and evaluate non-functional properties throughout QED's development. In particular, QED is using the *Component Workload Emulator (CoWorkEr) Utilization Test Suite* (CUTS) [10], which is a platform-independent SEM tool for component-based distributed systems. System testers use CUTS by modeling the behavior and workload of their component-based distributed system and generating a test system for their target architecture. System testers then execute the test system on their target architecture, and CUTS collects performance metrics, which can be used to unit test non-functional concerns. This process is repeated continuously throughout the software development lifecycle of the system under development.

Our prior work [8] showed how integrating CUTS-based SEM tools with continuous integtration environments provided a flexible solution for executing and managing component-based distributed system tests continuously throughout the development lifecycle. This prior work also showed how simple log messages can capture metrics of interest to evaluate non-functional concerns continuously throughout a system development. Applying the results of our prior work to the initial prototype of the QED middleware, however, revealed the following limitations of the initial version of CUTS:

• **Limitation 1: Inability to extract data for metrics of interest.** Data extraction is the process of locating relevant information in a data source that can be used for analysis. In the initial version of CUTS, data extraction was limited to metrics that CUTS knew *a priori*, *e.g.*, at compilation time. It was therefore hard to identify, locate, and extract data for metrics of interest, especially if a non-functional unit test needed data that CUTS did not know *a priori*.

QED testers needed a technique to identify metrics of interest that can be extracted from large amounts of system data. Moreover, the extraction technique should allow testers to identify key metrics at a high-level of abstraction and be flexible enough to handle data variation to apply CUTS effectively to large-scale systems.

• **Limitation 2: Inability to analyze and aggregate extracted data.** Data analysis and aggregation is the process of evaluating extracted data based on a user-defined equation, and combining multiple results (if applicable) to a single result. This process is necessary since unit testing operates on a single result—either simple or complex—to determine whether it passes or fails. In the initial version of CUTS, data analysis and aggregation was limited to functions that CUTS knew *a priori*, which made it hard to analyze extracted data via user-defined functions.

The QED testers need a flexible technique for collecting metrics that can be used in user-defined functions to evaluate various system-wide non-functional concerns, such as relative server utilization or end-to-end response time for events with different priorities. Moreover, the technique should preserve data integrity (*i.e.*, ensuring data is associated with the execution trace that generated it), especially in absence of a globally unique identifier to identify the correct execution trace that generated it.

Due to these limitations, it was hard for system developers to use the initial version of CUTS to conduct unit tests of non-functional concerns of QED. Moreover, this problem extends beyond the QED project and applies to other distributed systems that want to perform unit testing of non-functional concerns. The remainder of this paper shows how we addressed these limitations by improving CUTS so it could unit test non-functional concerns of component-based distributed systems throughout the software lifecycle.

3 UNITE: High-level Specification and Auto-analysis of Non-Functional Concerns

This section presents the underlying theory of UNITE and describes how it can be used to unit test non-functional concerns of component-based distributed systems.

3.1 Specification and Extraction of Metrics from Text-based System Logs

System logs (or execution traces) are essential to understanding the behavior of a system, whether or not the system is distributed [13]. Such logs typically contain key data that can be used to analyze the system online and/or offline. For example, Listing 1 shows a simple log produced a system.

```
1  activating LoginComponent
2  ...
3  LoginComponent recv request 6 at 1234945638
4  validating username and password for request 6
5  username and password is valid
6  granting access at 1234945652 to request 6
7  ...
8  deactivating the LoginComponent
```

Listing 1. Example log (or trace) produced by a system

As shown in Listing 1, each line in the log represents a system effect that generated the log entry. Moreover, each line captures the state of the system when the entry was produced. For example, line 3 states when a login request was received by the `LoginComponent` and line 6 captures when access was granted to the client by the `LoginComponent`.

Although a system log contains key data to analyzing the system that produced it, the log is typically generated in a verbose format that can be understood by humans. This format implies that most data is discardable. Moreover, each entry is constructed from a well-defined format, which we call a *log format*, that will not change throughout the lifetime of system execution. Instead, certain values (or variables) in each log format, such as time or event count, will change over the lifetime of the system. We formally define a log format $LF = (V)$ as:

- A set V of variables (or tags) that capture data of interest in a log message.

Moreover, Equation 1 determines the set of variables in a given log format LF_i.

$$V = vars(LF_i) \tag{1}$$

Implementing log formats in UNITE. To realize log formats and Equation 1 in UNITE, we use high-level constructs to identify variables $v \in V$ that contain data for analyzing the system. Users specify their message of interest and use placeholders—identified by brackets { }—to tag variables (or data) that can be extracted from an entry. Each placeholder represents variable portion of the message that may change over the course of the systems lifetime, thereby addressing Limitation 1 stated in Section 2. Table 1 lists the different placeholder types currently supported by UNITE. Finally, UNITE caches the variables and converts the high-level construct into a regular expression. The regular expression is used during the analysis process (see Section 3.3) to identify messages that have candidate data for variables V in log format LF.

Table 1. Log format variable types supported by UNITE

Type	Description
INT	Integer data type
STRING	String data type (with no spaces)
FLOAT	Floating-point data type

```
LF1: {STRING owner} recv request {INT reqid} at {INT recv}
LF2: granting access at {INT reply} to request {INT reqid}
```

Listing 2. Example log formats for tag metrics of interest

Listing 2 exemplifies high-level constructs for two log entries from Listing 1. The first log format (LF_1) is used to locate entries related to receiving a login request for a client (line 3 in Listing 1). The second log format (LF_2) is used to locate entries related to granting access to a client's request (line 6 in Listing 1). Overall, there are 5 tags in Listing 2. Only two tags, however, capture metrics of interest: `recv` in LF_1 and `reply` in LF_2. The remaining three tags (*i.e.*, `owner`, `LF1.reqid`, and `LF2.reqid`) are used to preserve causality, which we explain in more detail in Section 3.2.

3.2 Specification of Unit Test for Analyzing Non-functional Concerns

Section 3.1 disussed how we use log formats to identify entries in a log that contain data of interest. Each log format contains a set of tags, which are representative of variables and used to extract data from each format. In the trivial case, a single log format can be used to analyze a non-functional concern. For example, if a developer wanted to know how many events a component received per second, then the component could cache the necessary information internally and generate a single log message when the system is shutdown.

Although this approach is feasible, *i.e.*, caching data and generating a single message, it is not practical in a component-based distributed system because individual data points used to analyze the system can be generated by different components. Moreover, data points can be generated from components deployed on different hosts. Instead, what is needed is the ability to generate independent log messages and specify how to associate the messages with each other to preserve data integrity.

In the context of unit testing non-functional concerns, we formally define a unit test $UT = (LF, CR, f)$ as:

- A set LF of log formats that have variables V identifying which data to extract from log messages.
- 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.
- A user-defined evaluation function f based on the variables in LF.

Causal relations are traditionally based on time. UNITE, however, uses log format variables to resolve causality because it alleviates the need for a globally unique identifier to associate metrics (or data). Instead, you only need to ensure that two unique log formats can be associated with each other, and each log format is in at least one causal relation (or association). UNITE does not permit circular relations because it requires human feedback to determine where the relation chain between log formats begins and ends.

We formally define a causal relation $CR_{i,j} = (C_i, E_j)$ as:

- A set $C_i \subseteq vars(LF_i)$ of variables that define the key to represent the cause of the relation.
- A set $E_j \subseteq vars(LF_j)$ of variables that define the key to represent the effect of the relation.

Moreover, $|C_i| = |E_j|$ and the type of each variable (see Table 1), *i.e.*, $type(v)$, in C_i, E_j is governed by Equation 2:

$$type(C_{i_n}) = type(E_{j_n}) \tag{2}$$

where $C_{i_n} \in C_i$ and $E_{j_n} \in E_j$.

Implementing unit tests in UNITE. In UNITE, users define unit tests by selecting what log formats should be used to extract data from message logs. If a unit test has more than one log format, then users must create a causal relation between each log format. When specifying casual relations, users select variables from the corresponding log format that represent the cause and effect. Last, users define an evaluation function based on the variables in selected log formats.

For example, if a QED developer wanted to create a unit test to calculate duration of the login operation, then a unit test is created using LF_1 and LF_2 from Listing 2. Next, a causal relation is defined between LF_1 and LF_2 as:

$$LF_1.reqid = LF_2.reqid \tag{3}$$

Finally, the evaluation function is defined as:

$$LF_2.reply - LF_1.recv \tag{4}$$

The following section discusses how we evaluate the function of non-functional unit tests for component-based distributed systems.

3.3 Evaluation of Non-functional Unit Tests

Section 3.1 discussed how log formats can be used to identify messages that contains data of interest. Section 3.2 discussed how to use log formats and casual relations to specify unit test for non-functional concerns. The final phase of the process is evaluating the unit test, *i.e.*, the evaluation function f. Before we explain the algorithm used to evaluate a unit test's function, we must first understand different types of causal relations that can occur in a component-based distributed system.

As shown in Figure 3, there are four types of causal relations that can occur in a component-based distributed system, which affect the algorithm used to evaluate a unit test. The first type (a) is one-to-one relation, which is the most trivial type to resolve between multiple log formats. The second type (b) is one-to-many relation and is a result of a multicast event. The third type (c) is many-to-one, which occurs when many different components send a event type to a single component. The final type (d) is a combination of previous types (a) – (d), and is the most complex relation to resolve between multiple log formats.

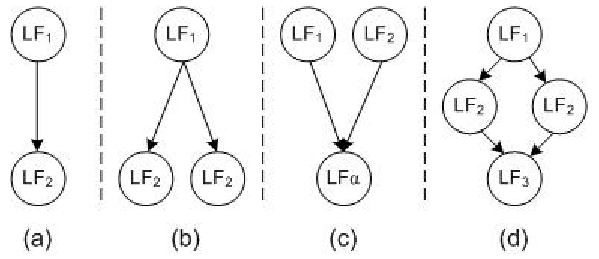

Figure 3. Four Types of Causal Relations That Can Occur in a Component-based Distributed System

If we assume that each entry in a message log contains its origin, *e.g.*, hostname, then we can use dynamic programming algorithm and relational database theory to reconstruct the data table of values for a unit test's variables. Algorithm 1 lists our algorithm for evaluating a unit test. As shown in Algorithm 1, we evaluate a unit test UT given the collected log messages LM of the system. The first step to the evaluation process is to create a directed graph G where log formats LF are nodes and the casual relations $CR_{i,j}$ are

Algorithm 1 General algorithm evaluating a unit test via log formats and causal relations

```
 1: procedure EVALUATE($UT$, $LM$)
 2:     $UT$: unit test to evaluate
 3:     $LM$: set of log messages with data
 4:     $G \leftarrow directed\_graph(UT)$
 5:     $LF' \leftarrow topological\_sort(G)$
 6:     $DS \leftarrow variable\_table(UT)$
 7:     $LM' \leftarrow$ sort $LM$ ascending by (origin, time)
 8:
 9:     for all $LF_i \in LF'$ do
10:         $K \leftarrow C_i$ from $CR_{i,j}$
11:
12:         for all $LM_i \in LM'$ do
13:             if $matches(LF_i, LM_i)$ then
14:                 $V' \leftarrow$ values of variables in $LM_i$
15:
16:                 if $K \neq \emptyset$ then
17:                     $R \leftarrow findrows(DS, K, V')$
18:                     $update(R, V')$
19:                 else
20:                     $append(DS, V')$
21:                 end if
22:             end if
23:         end for
24:     end for
25:
26:     $DS' \leftarrow$ purge incomplete rows from $DS$
27:     return $f(DS')$ where $f$ is evaluation function for $UT$
28: end procedure
```

edges. We then topologically sort the directed graph so we know the order to process each log format. This step is necessary because when causal relation types (b) – (d) are in the unit test specification, processing the log formats in reverse order of occurrence reduces algorithm complexity for constructing data set DS. Moreover, it ensures we have rows in the data set to accommodate the data from log formats that occur prior to the current log format. After topologically sorting the log formats, we construct a data set DS, which is a table that has a column for each variable in the log formats of the unit test.

Likewise, we sort the log messages by origin and time to ensure we have the correct message sequence for each origin. This step is necessary if you want to see data trend over the lifetime of the system before aggregating the results, which we discuss later in the paper. Once we have sorted the log messages, we match each log format in LF' against each log message in LM'. If there is a match, then we extract values of each variable from the log message, and update the data set. If there is a cause variable set C_i for the log format LF_i, we locate all the rows in the data set where the values of C_i equal the values of E_j, which are set by processing the previous log format. If there is no cause variable set, we append the values from the log message to the end of the data set. Finally, we purge all the incomplete rows from the data set and evaluate the data set using the user-defined evaluation function for the unit test.

Handling duplicate data entries. For long running systems, it is not uncommon to see variations of the same log message within the complete set of log messages. Moreover, we defined log formats on a unit test to identify variable portions of a message (see Section 3.1). We therefore expect to encounter the same log format multiple times. When constructing the data set in Algorithm 1, different variations of the same log format will create multiple rows in final data set. A unit test, however, operates on a single value, and not multiple values. To address this concern, we use the following techniques:

- **Aggregation.** A function used to convert a data set to a single value. Examples of an aggregation function are, but not limited to: AVERAGE, MIN, MAX, and SUM.
- **Grouping.** Given an aggregation function, grouping is used to identify data sets that should be treated independent of each other. For example, in the case of causal relation (d) in Figure 3, the values in the data set for each sender (*i.e.*, LF_2) could be considered a group and analyzed independently.

We require specifying of an aggregation function as part of the evaluation equation f for a unit test because it is known *a priori* if a unit test may produce a data set with multiple values. We formally define a unit test with groupings $UT' = (UT, \Gamma)$ as:

- A unit test UT for evaluating a non-funtional concern.
- A set $\Gamma \subseteq vars(UT)$ of variables from the log formats in the unit test.

Evaluating unit tests in UNITE. In UNITE, we implemented Algorithm 1 using the SQLite relational database (`sqlite.org`). To construct the variable table, we first insert the data values for the first log format directly into the table since it has no causal relations. For the remaining log formats, we transform its causal relation(s) into a SQL `UPDATE` query. This allows us to update only rows in the table where the relation equals values of interest in the current log message. Table 2 shows the variable table constructed by UNITE for the example unit test in Section 3.2. After the variable data table is constructed, we use the evaluation function and groupings for the unit test to create the final SQL query that evaluates the unit test.

Table 2. Example Data Set Produced from Log Messages

LF1_reqid	LF1_recv	LF2_reqid	LF2_reply
6	1234945638	6	1234945652
7	1234945690	7	1234945705
8	1234945730	8	1234945750

```
SELECT AVERAGE (LF2_reply - LF1_recv) AS result
  FROM vtable123;
```

Listing 3. SQL query for calculation average login duration

Listing 3 shows Equation 4 as an SQL query, which is used to evaluate the data set in Table 2. The final result of this example—and the unit test—would be 16.33 msec.

4 Applying UNITE to the QED Project

This section analyzes results of experiments that evaluate how UNITE can address key testing challenges of the QED project described in Section 2.

4.1 Experiment Setup

As mentioned in Section 2, the QED project is in its first year of development and is expected to continue for several years. QED developers do not want to wait until system integration time to validate the performance of their middleware infrastructure relative to stated QoS requirements. QED testers therefore used CUTS [10] and UNITE to perform early intergration testing. All tests were run in a representative testbed based on ISISlab (`www.isislab.`

`vanderbilt.edu`), which is powered by Emulab software [20][1]. Each host in our experiment was an IBM Blade Type L20, dual-CPU 2.8 GHz processor with 1 GB RAM configured with the Fedora Core 6 operating system.

To test the QED middleware, we first constructed several scenarios using CUTS' modeling languages [9]. Each scenario was designed such that all components communicate with each other using a single server in the GIG (similar to Figure 2 in Section 2). The first scenario was designed to test different thresholds of the underlying GIG middleware to pinpoint potential areas that could be improved by the QED middleware. The second scenario was more complex and emulated a *multi-stage workflow*. The multi-stage workflow is designed to test the underlying middleware's ability to ensure application-level QoS properties, such as reliability and end-to-end response time when handling applications with different priorities and priviledges.

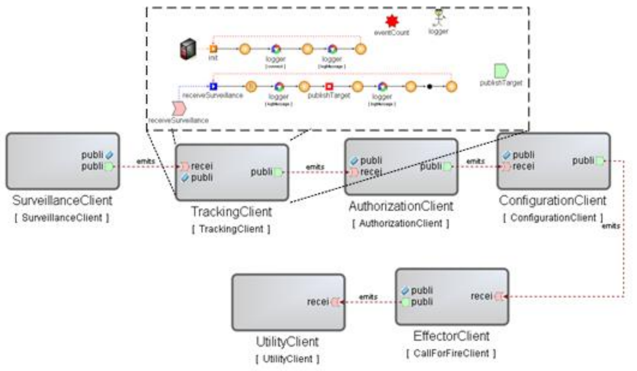

Figure 4. CUTS model of the multi-stage workflow test scenario

As shown in Figure 4, the multi-stage workflow has six types of components. Each directed line that connects a component represents a communication event (or stage) that must pass through the GIG (and QED) middleware before being delivered to the component on the opposite end. Moreover, each directed line conceptually represents where QED will be applied to ensure QoS between communicating components.

The projection from the middle component represents the behavior of that specific component. Each component in the multi-stage workflow has a behavior model (based on Timed I/O Automata [9]) that dictates its actions during a test. Moreover, each behavior model contains actions for logging key data needed to evaluate a unit test, similar to Listing 1 in Section 3.1.

Listing 4 lists an example message from the multi-stage workflow scenario.

```
. MainAssembly.SurveillanceClient: Event 0: Published a
  SurveillanceMio at 1219789376684
. MainAssembly.SurveillanceClient: Event 1: Time to
  publish a SurveillanceMio at 1219789376685
```

Listing 4. Example log messages from the multi-stage workflow scenario

This log message contains information about the event, such as event id and timestamp. Each component also generates log messages about the events it receives and its state (such as event count). In addition, each component sends enough information to create a causal relation between itself and the receiver, so there is no need for a global unique identifier to correlate data.

We next used UNITE to construct log formats (see Section 3.1) for identifing log messages during a test run that contain metrics of interest. These log formats were also used to define unit tests that evaluate non-functional concerns described in Section 3.2. Overall, we were interested in evaluating the following concerns in our experiments:

• **Multiple publishers.** At any point in time, the GIG will have many components publishing and receiving events simultaneously. We therefore need to evaluate the response time of events under such operating conditions. Moreover, QED needs to ensure QoS when the infrastructure servers must manage many events. Since the QED project is still in the early stages of development, we must first understand the current capabilities of the GIG middleware without QED in place. These results provide a baseline for evaluating the extent to which the QED middleware capabilities improve application-level QoS.

• **Time spent in server.** One way to ensure high QoS for events is to reduce the time an event spends in a server. Since the GIG middleware is provided by a third-party vender, we cannot ensure it will generate log messages that can be used to calculate how it takes the server to process an event. Instead, we must rely on messages generated from distributed application components whenever it publishes/-sends an event.

For an event that propogates through the system, we use Equation 5 to calculate how much time the event spends in the server assuming event transmission is instantenous, *i.e.*, neglictable.

$$(end_e - start_e) - \sum_c S_{c_e} \qquad (5)$$

As listed in Equation 5, we calculate the time spent in the server by taking the response time of the event e, and subtracting the sum of the service time of the event in each component S_{c_e}. Again, since QED is in its early stages of development, this unit test will be provide a baseline of

[1]Emulab allows developers and testers to configure network topologies and operating systems on-the-fly to produce a realistic operating environment for distributed unit and integration testing.

the infrastructure and used continuously throughout development.

4.2 Experiment Results

We now present results for experiments of the scenarios discussed in Section 4.1. Since QED is still in the early stages of development, we focus our experiments on evaluating the current state of the GIG middleware infrastructure, *i.e.*, without measuring the impact of QED middleware on QoS, and UNITE's ability to unit test non-functional concerns.

Analyzing multiple publisher results. Table 3 shows the results for the unit test that evaluates average end-to-end response time for an event when each publisher publishes at 75 Hz. As expected, the response time for each importance value was similar. When we unit tested this scenario using UNITE, the test results presented in Table 3 were calculated from two different log formats—either log format generated by a publisher and the subscriber. The total number of log messages generated during the course of the test was 993,493.

Table 3. Average end-to-end (E2E) response time (RT) for multiple publishers sending events at 75 Hz

Publisher Name	Importance	Avg. E2E RT (msec)
ClientA	30	103931.14
ClientB	15	103885.47
ClientC	1	103938.33

UNITE also allows us to view the data trend for the unit test of this scenario to get a more detailed understanding of performance. Figure 5 shows how the response time of the event increases over the lifetime of the experiment. We knew beforehand that the this configuration for the unit test produced too much workload. UNITE's data trend and visualazation capabilities, however, helped make it clear the extent to which the GIG middleware was being over utilized.

Analyzing maximum sustainable publish rate results. We used the multi-stage workflow to describe a complex scenario tests the limits of the GIG middleware without forcing it into incremental queueing of events. Figure 6 graphs the data trend for the unit test, which is calculated by specifying Equation 5 as the evaluation for the unit test, and was produced by UNITE after analyzing (*i.e.*, identifying and extracting metrics from) 193,464 log messages. The unit test also consisted of ten different log formats and nine different causal relations, which were of types (a) and (b), discussed in Section 3.3.

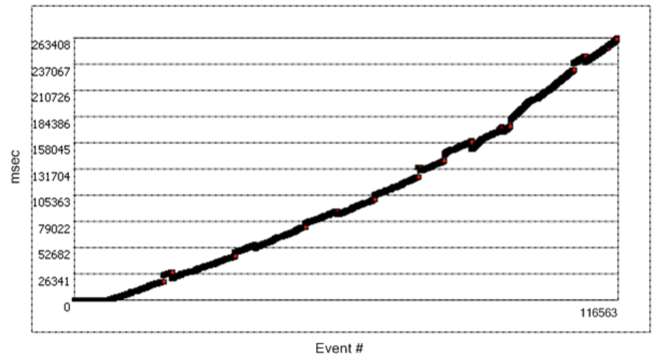

Figure 5. Data trend graph of average end-to-end response time for multiple publishers sending events at 75 Hz

Figure 6 illustrates the sustainable publish rate of the mutle-stage workflow in ISISlab. As illustrated, the just-in-time compiler (JITC) and other Java features cause the the middleware to temporarily increase the individual message end-to-end response. By the end of the test (which is not shown in the above graph), the time an event spends in the server reduces to normal operating conditions.

The multi-stage workflow results provided two insights to QED developers. First, their theory of maximum publish rate in ISISlab was confirmed. Second, Figure 6 helped developers speculate on what features of the GIG middleware might cause performance bottlenecks, how QED could address such problems, and what new unit test are need to illustrate QED's improvements to the GIG middleware. By providing QED testers comprehensive testing and analysis features, UNITE helped guide the development team to the next phase of testing and integration of feature sets.

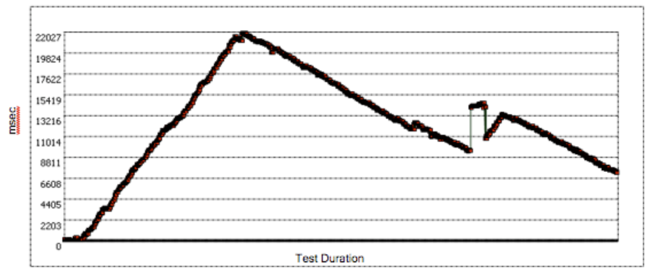

Figure 6. Data trend of the system placed in near optimal publish rate

Evaluating the impact of UNITE on the experiment. Because of UNITE, we could quickly construct unit tests to evaluate the GIG middleware. In context of the multi-stage workfow scenario, UNITE provided two insights to QED developers. First, their theory of maximum publish rate in ISISlab was confirmed. Second, the data trend and visualization capabilities of UNITE helped developers speculate on what features of the GIG middleware might cause performance bottlenecks, how QED could address such problems, and what new unit test are need to illustrate QED's improvements to the GIG middleware.

In addition, UNITE's analyical capabilities are not bounded by a unit test's configuration and data. As long as the correct log formats and their causal relations is specified, UNITE can evaluate the unit test. Moreover, we did not need to specify a global unique identify to associate data with its correct exeuction trace. If we required a global unique identifier to associate data metrics, then we would have to ensure that all components propogated the identifer. Moreover, if we added new compoents to the multi-stage workflow, each component would have to be aware of the global unique idenfier, which can inherently complicate the logging specification.

By providing developers comprehensive testing and analysis capabilities, UNITE helped guide the QED developers to the next phase of testing and integration of feature sets. We therefore conclude that UNITE helped reduce the complexity of evaluating a non-functional concerns of a component-based distributed system. Moreover, unit tests can be automated and run continuously throughout the software lifecycle of QED using CiCUTS [8], which integrates CUTS with continuous integration environments, such as CruiseControl (`cruisecontrol.sourceforge.net`).

5 Related Work

This section compares our work on UNITE with related work on unit testing and component-based distributed system analysis.

Distributed system unit testing. Coelho et al. [4] and Yamany et. al [24] describe techniques for unit testing multi-agent systems using so-called mock objects. Their goal for unit testing multi-agent systems is similar to UNITE, though they focus on functional concerns, whereas UNITE focuses on non-functional concerns of a distributed system during the early stages of development. Moreover, Coelho et. al unit test a single multi-agent isolation, whereas UNITE focuses on unit testing systemic properties (*i.e.*, many components working together). UNITE can also be used to unit test a component in isolation, if necessary.

Qu et. al [19] present a tool named *DisUnit* that extends JUnit [16] to enable unit testing of component-based distributed systems. Although DisUnit supports testing of distributed systems, it assumes that metrics used to evaluate a non-functional concern are produced by a single component. As a result, DisUnit cannot be used to test non-funtional concerns of a distributed system where metrics are dispersed throughout a system execution trace, which can span many components and hosts in the system. In contrast, UNITE assumes that data need to evaluate a test can occur in any location and at any time during the system's execution.

Component-based distributed systems analysis. Mania et. al [15] discuss a technique for developing performance models and analyzing component-based distributed system using execution traces. The contents of traces are generated by system events, similar to the log message in UNITE. When analyzing the systems performance, however, Mania et. al rely on synchronized clocks to reconstruct system behavior. Although this technique suffices in tightly coupled environments, if clocks on different hosts drift (as may be the case in ultra-large-scale systems), then the reconstructed behavior and analysis may be incorrect. UNITE improves upon their technique by using data within the event trace that is common in both cause and effect messages, thereby removing the need for synchronized clocks and ensuring that log messages (or events in a trace) are associated correctly.

Similarly, Mos et al. [17] present a technique for monitoring Java-based components in a distributed system using proxies, which relies on timestamps in the events and implies a global unique identifier to reconstruct method invocation traces for system analysis. UNITE improves upon their technique by using data that is the same between two log messages (or events) to reconstruct system traces given the causal relations between two log formats. Moreover, UNITE relaxes the need for a global identifier.

Parsons et al. [18] present a technique for performing end-to-end event tracing in component-based distributed systems. Their technique injects a global unique identifier at the beginning of the event's trace (*e.g.*, when a new user enters the system). This unique identifier is then propagated through the system and used to associate data for analytical purposes. UNITE improves upon their technique by relaxing the need for a global unique identifier to associate data for analysis. Moreover, in a large- or ultra-large-scale component-based distributed system, it can be hard to ensure unique identifiers are propagated throughout components created by third parties.

6 Concluding Remarks

Non-functional concerns of component-based distributed systems have traditionally been tested during integration. The earlier that non-functional concerns are tested in the actual target environment, however, the greater chance of locating problematic areas in the software system [22, 23]. This paper describes and evaluates a technique called *Understanding Non-functional Intentions via*

Testing and Experimentation (UNITE) for unit testing non-functional concerns of component-based distributed systems. UNITE uses log messages generated from a testing environment and message constructs that identify messages of interest. Moreover, testers define functions in terms of variable data in log messages that are used to evaluate non-functional concerns of the system under development.

Our experience applying UNITE to a representative component-based distributed system showed how it simplified identifying and extracting of metrics of interest for analyzing non-functional concerns. Moreover, UNITE did not require us to use a global unique identfier to reconstruct system traces, thereby reducing the complexity of generated log messages. As shown in this paper, UNITE reduces the inherit complexities of unit testing non-functional concerns of a component-based distributed system during the early stages of development. It is currently unable, however, to analyze a subset of the extracted metrics based on parameterizing the causal relations. We plan to add this support in future work.

CUTS and UNITE are freely available in open-source format at `www.dre.vanderbilt.edu/CUTS`.

References

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

[2] P. Atzeni and V. D. Antonellis. *Relational Database Theory*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[3] D. Box and D. Shukla. WinFX Workflow: Simplify Development with the Declarative Model of Windows Workflow Foundation. *MSDN Magazine*, 21:54–62, 2006.

[4] R. Coelho, U. Kulesza, A. von Staa, and C. Lucena. Unit Testing in Multi-agent Systems using Mock Agents and Aspects. In *International Workshop on Software Engineering for Large-scale Multi-agent Systems*, pages 83–90, 2006.

[5] G. Denaro, A. Polini, and W. Emmerich. Early Performance Testing of Distributed Software Applications. *ACM SIGSOFT Software Engineering Notes*, 29(1):94–103, January 2004.

[6] M. Dutoo and F. Lautenbacher. Java Workflow Tooling (JWT) Creation Review. www.eclipse.org/proposals/jwt/ JWT%20Creation%20Review%2020070117.pdf, 2007.

[7] M. El-Gendy, A. Bose, and K. Shin. Evolution of the internet qos and support for soft real-time applications. *Proceedings of the IEEE*, 91(7):1086–1104, July 2003.

[8] J. Hill, D. C. Schmidt, J. Slaby, and A. Porter. CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments. In *Proceeedings of 15th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*, Belfast, Northern Ireland, March 2008.

[9] J. H. Hill and A. Gokhale. Model-driven Engineering for Early QoS Validation of Component-based Software Systems. *Journal of Software (JSW)*, 2(3):9–18, Sept. 2007.

[10] 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.

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

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

[13] N. Joukov, T. Wong, and E. Zadok. Accurate and Efficient Replaying of File System Traces. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 25–25, 2005.

[14] J. Loyall, M. Carvalho, D. Schmidt, M. Gillen, A. M. III, L. Bunch, J. Edmondson, and D. Corman. QoS Enabled Dissemination of Managed Information Objects in a Publish-Subscribe-Query Information Broker. In *Defense Transformation and Net-Centric Systems*, April 2009.

[15] D. Mania, J. Murphy, and J. McManis. Developing Performance Models from Nonintrusive Monitoring Traces. *IT&T*, 2002.

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

[17] A. Mos and J. Murphy. Performance Monitoring of Java Component-Oriented Distributed Applications. In *IEEE 9th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 9–12, 2001.

[18] T. Parsons, Adrian, and J. Murphy. Non-Intrusive End-to-End Runtime Path Tracing for J2EE Systems. *IEEE Proceedings Software*, 153:149–161, August 2006.

[19] R. Qu, S. Hirano, T. Ohkawa, T. Kubota, and R. Nicolescu. Distributed Unit Testing. Technical Report CITR-TR-191, University of Auckland, 2006.

[20] 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.

[21] M. Tortonesi, C. Stefanelli, N. Suri, M. Arguedas, and M. Breedy. Mockets: A Novel Message-Oriented Communications Middleware for the Wireless Internet. In *International Conference on Wireless Information Networks and Systems (WINSYS 2006)*, August 2006.

[22] E. J. Weyuker. Testing Component-based Software: A Cautionary Tale. *Software, IEEE*, 15(5):54–59, Sep/Oct 1998.

[23] Y. Wu, M.-H. Chen, and J. Offutt. UML-Based Integration Testing for Component-Based Software. In *Proceedings of the Second International Conference on COTS-Based Software Systems*, pages 251–260. Springer-Verlag, 2003.

[24] H. F. E. Yamany, M. A. M. Capretz, and L. F. Capretz. A Multi-Agent Framework for Testing Distributed Systems. In *30th Annual International Computer Software and Applications Conference*, pages 151–156, 2006.