Validating Dynamic Enterprise Distributed Real-time and Embedded System Quality-of-Service Properties using System Execution Traces

James H. Hill[a,*], Pooja Varshneya[b], Hamilton A. Turner[b], James R. Edmondson[b], Douglas C. Schmidt[b]

[a] *Dept. of Computer and Information Science, Indiana University-Purdue University Indianapolis, Indianapolis, IN 46202, USA*
[b] *Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37204, USA*

Abstract

System execution traces can be used to validate enterprise distributed real-time and embedded (DRE) systems quality-of-service (QoS) properties (such as response-time, latency, and scalability) based on system structure/composition. As enterprise DRE systems increase in size (*i.e.*, number of hardware/-software components) and complexity (*i.e.*, envisioned operational scenarios) it becomes harder to validate QoS properties because traditional techniques do not adapt to the dynamic nature of the system. This article therefore describes a methodology and tool called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)* that uses relational database techniques and dataflow models to validate enterprise DRE system's QoS properties independent of the system's structure/composition. Our empirical results show that UNITE is a lightweight and scalable method whose evaluation time depends primarily on the amount of data being analyzed, as opposed to the size, composition, and complexity of the enterprise DRE system itself.

Keywords: Enterprise distributed real-time and embedded (DRE) systems, Dynamic systems, Quality-of-service validation, System execution traces, Relational database theory

1. Introduction

Emerging trends and challenges. Enterprise DRE systems (*e.g.*, urban traffic management systems, air traffic control systems, powergrid SCADA sys-

*Corresponding author
Email addresses: hillj@cs.iupui.edu (James H. Hill), pooja.varshneya@vanderbilt.edu (Pooja Varshneya), hamilton.a.turner@vanderbilt.edu (Hamilton A. Turner), james.r.edmondson@vanderbilt.edu (James R. Edmondson), d.schmidt@vanderbilt.edu (Douglas C. Schmidt)

tems, and shipboard computing systems) are a class of systems that must satisfy functional (*e.g.*, operational capabilities) and quality-of-service (QoS) requirements (*e.g.*, end-to-end response time, throughput, and scalability) [1]. *System execution traces*, which are a collection of messages that reflect events that occur throughout the execution lifetime of a system, can be used to validate enterprise DRE system's functional [2, 3] and QoS properties [4, 5, 6]. Enterprise DRE system developers can use these traces to determine if system functionality executes correctly by validating that the execution trace either (1) contains a message (or messages) that reflects execution of the functional concern or (2) does not contain an error message that reflects failure related to the functional concern. This process can often be automated to reduce the amount of manual time and effort needed to validate functional properties using system execution traces [3].

Validate QoS properties with system execution traces is often harder than validating functional properties since the data points needed to validate individual QoS properties can be dispersed throughout a system execution trace. For example, validating end-to-end response time of an event requires the event's timestamp at its source and destination while tracking (or correlating) the event across the entire system. The event's initial timestamp usually appears earlier than its final timestamp in the system execution trace. A common approach to performing this correlation is to send as much data as possible (*e.g.*, an event id and initial timestamp) to perform validation at the final destination (*e.g.*, subtract the initial timestamp from the final timestamp). Although this approach does not require correlating data points, it can be expensive in terms of resources used (*e.g.*, CPU, memory, and network bandwidth) and negatively impact existing functional and QoS properties.

Other approaches for validating QoS properties using system execution traces are tightly coupled to (1) *system implementation* [4, 6], *i.e.*, what technologies are used to implement the system, and (2) *system composition* [5, 6], *i.e.*, where components are located and what components communicate with each other. Although these approaches are feasible, their constraints often limit the validation of QoS properties in enterprise DRE system —particularly *dynamic* systems—since individual snapshots of system composition and structure are needed each time it dynamically changes (*e.g.*, addition/removal of a component and moving a component to a new host) to perform the validation. New techniques are therefore needed to improve the QoS validation capabilities of dynamic enterprise DRE systems so it is not tightly coupled to system composition and system implementation.

Solution approach → QoS validation with dataflow models. To validate QoS properties independent of system composition and implementation, the validation process should be performed using abstractions that remain (relatively) constant across system composition and implementation. An abstraction that meets this criteria is a *dataflow model* [7], which describe how data is transmitted through an information system, because a dataflow often remains constant, even as system compositions and implementations change. For example, dataflow models have been used in UML to capture information

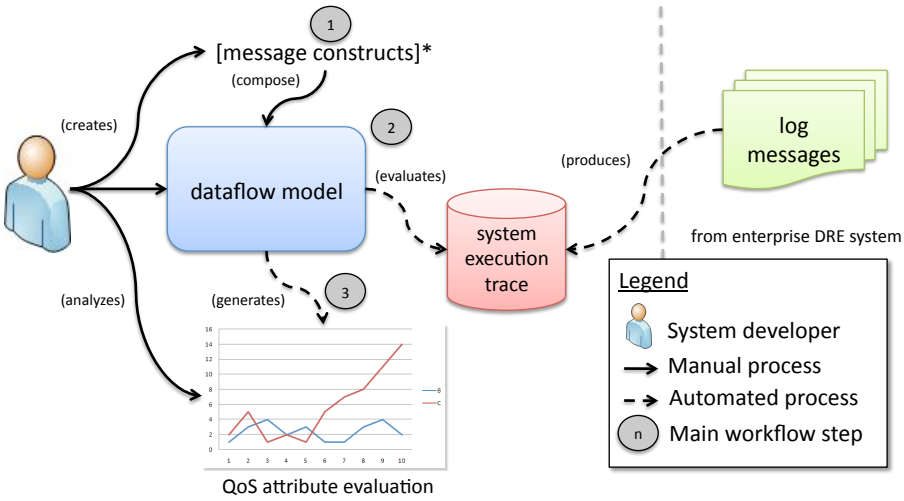

Figure 1: Overview of UNITE's Workflow

flow in software systems independent of its composition and implementation [8]. In the context of enterprise DRE systems, dataflow models describes how data (1) flows between different components distributed across hosts in the target environment and (2) is exchanged via interprocess communication (IPC) mechanisms, such as distributed objects, publish/subscribe, and messaging. These models can then be used in conjunction with system execution traces to validate enterprise DRE system QoS properties.

This article describes a method and tool called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)* that uses dataflow models to validate enterprise DRE system QoS properties via system execution traces. UNITE analyzes dataflow models using relational database theory techniques [9], such as table relations and joins, where the correlation of data points used to validate a QoS property are associated with each other via their relations in the dataflow model. The resultant data table is then evaluated by applying an SQL expression based on a user-defined function.

Enterprise DRE system developers and testers can use UNITE to validate enterprise DRE system QoS properties via the steps shown in Figure 1 and summarized below:

1. Identity data of interest within the messages using message constructs, *e.g.*,: `{STRING ident} sent message {INT eventId} at {INT time}`;
2. Define a dataflow model for extracting metrics of interest used in QoS validation; and
3. Define a QoS validation equation to analyze dataflow model and evaluate a QoS property, such as end-to-end response time, latency, and scalability.

Our experience of applying UNITE to a representative enterprise DRE system shows it is an effective technique for validating QoS properties independent

of system composition and implementation details. Moreover, our results show that although evaluation time increases as the size of UNITE dataflow models increase (which is correlated with system size and complexity), evaluation time depends primarily on the amount of data being analyzed. UNITE can therefore scale up to handle large DRE systems consisting of many components and hosts that contain complex behavior without unduly degrading evaluation time of QoS properties using system execution traces.

Article organization. The remainder of this article is organized as follows: Section 2 summarizes a representative DRE system case study to motivate the challenges addresed by UNITE; Section 3 describes the structure and functionality of UNITE and shows how UNITE addresses the challenges introduced in the motivating case study; Section 4 analyzes the results of experiments that evaluate UNITE in the context of 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 [10] is a enterprise DRE system from the class of ultra-large-scale (ULS) systems [11]. 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 QoS assurance to applications that must respond rapidly to priority shifts and unfolding situations.

The QoS-Enabled Dissemination (QED) [12] project is a multi-year, multi-organization collaboration designed to improve GIG middleware so it can meet QoS requirements of users and component-based distributed systems. QED 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

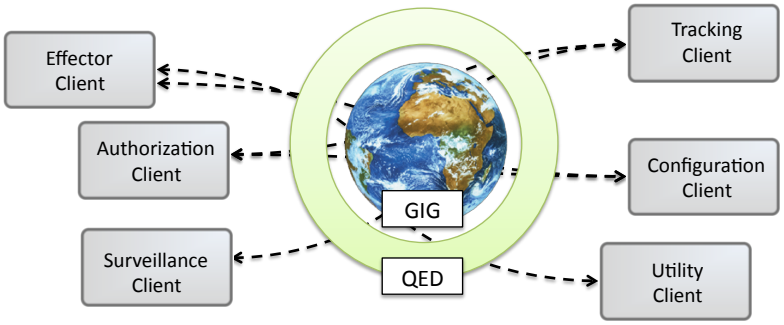

Figure 2: Conceptual Model of QED in the Context of the GIG

information broker based on 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 QoS-enabled network technologies (such as Mockets [13] and differentiated service queues [14]) to provide end-to-end QoS assurance to GIG applications.

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 the *serialized-phasing problem* [15], which is common in large-scale enterprise DRE systems. In the serialized-phasing problem, the system is developed in layers, where components in the upper layer(s) are not developed until (often long) after the components in the lower layer(s) are developed. Design flaws that affect QoS properties are thus often not discovered until final integration (*e.g.*, at system integration time), when they are more costly and harder to fix [16, 17].

To overcome the serialized-phasing problem, QED developers are using system execution modeling tools [18, 19, 20, 21] to execute performance regression tests automatically against the QED middleware and evaluate QoS properties continuously throughout its development. In particular, QED uses the *Component Workload Emulator (CoWorkEr) Utilization Test Suite* (CUTS) [19], which is a platform-independent system execution modeling tool for enterprise DRE systems. Enterprise DRE system developers and testers use CUTS by modeling the behavior and workload of their enterprise DRE system and generating a test system for their target architecture. These developers and testers then execute the test system on their target architecture, and CUTS collects performance metrics, which can be used to evaluate QoS properties. This process is repeated continuously throughout the software lifecycle to increase confidence in QoS assurance.

Prior work [22] showed how integrating CUTS with continuous integration environments provided a flexible solution for executing and managing component-based distributed system tests continuously throughout the development lifecycle. This work also confirmed that system execution traces can be used to validate enterprise DRE system QoS properties. Applying the results of prior work to the initial prototype of the QED middleware, however, revealed the following limitations with 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 QoS validation functions needed data that CUTS did not know *a priori*, such as metrics extracted from third-party components and CUTS is not aware of its implementation.

  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 operate independent of system composition and

implementation, and be flexible enough to apply effectively to dynamic enterprise DRE systems. Sections 3.2 and 3.3 describe how UNITE evaluates QoS properties within system execution traces using log formats and dataflow models to address this limitation with CUTS.

- **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 QoS validation traditionally yields a scalar value, such as average latency or worst case response time. In the initial version of CUTS, data analysis and aggregation was limited to functions that CUTS knew *a priori*, *i.e.*, built-in analytical equations. This made it hard to analyze extracted data via user-defined functions, and implied analysis was tightly coupled to system implementation and system composition, and the CUTS tool itself.

  QED testers need a flexible technique for collecting metrics that can be used in user-defined functions to evaluate various system-wide QoS properties, 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, such as a system-wide unique id associated with each piece of generated data, to identify the correct execution trace that generated it. Section 3.4 describes how UNITE evaluates dataflow models via relational database theory techniques to address this limitation with CUTS.

- **Limitation 3: Inability to manage complexity of QoS property evaluation specification.** As enterprise DRE systems increase in size and complexity, the challenges associated with limitations 1 and 2 described above can also increase in complexity. For example, as enterprise DRE system implementations mature, more components are often added and the amount of data generated for QoS property evaluation will increase. Likewise, the specification of a QoS property evaluation equations will also increase because there is more data to manage and filter.

  QED testers need a flexible and lightweight technique that will ensure complexities associated with limitations 1 and 2 are addressed properly as the QED implementation matures and increases in size and complexity. Moreover, the technique should enforce constraints of the overall process, but be intuitive to use so that QED testers can focus more on QoS property evaluation as opposed to specification of QoS property evaluation. Section 3.5 describes how UNITE uses domain-specific modeling languages to address this limitation with CUTS.

The limitations with CUTS described above made it hard for QED developers to use CUTS and related approaches to validate QoS properties using system execution traces without tight-coupling to both system implementation and system composition. Moreover, this problem extends beyond the QED project and

applies to other enterprise DRE systems that want validate QoS properties using system execution traces. The remainder of this article shows how UNITE addresses these limitations and improves CUTS analytical capabilities for assuring enterprise DRE system QoS properties.

3. UNITE: High-level QoS Evaluation using Dataflow Models

This section presents the structure and function of UNITE, focusing on it use of dataflow models to facilitate the implementation and composition-independent validation of enterprise DRE system's QoS properties.

3.1. Motiviation for Using Dataflow Models

Before describing UNITE's structure and functionality, it is first necessary to motivate its use of dataflow models. Section 1 outlined how dataflow models can be used to validate enterprise DRE system QoS properties independently from system implementation and composition. It is straightforward to understand how dataflow models can function independently of system implementation since they are higher level of abstraction than system implementation logic. Understanding how dataflow models can function independently of system composition, however, is more subtle.

Figure 3 shows two different deployments (*i.e.*, runtime compositions) of the same system. This figure shows 2 different component types (*i.e.*, `Sensor` and

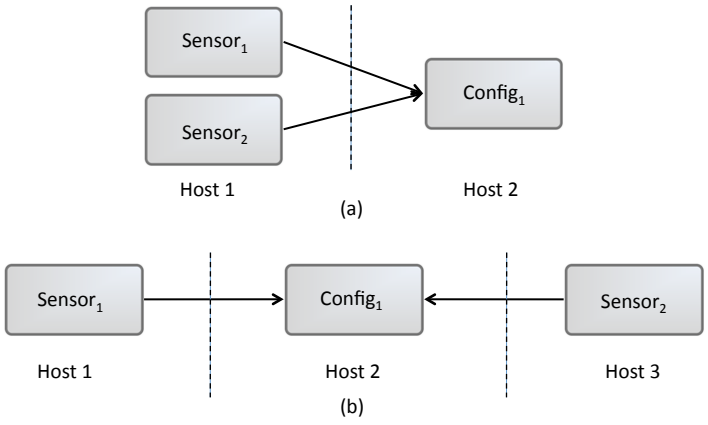

Figure 3: Example of Two Different Deployments for the Same System

`Config`) and 3 different component instances (*i.e.*, $\texttt{Sensor}_1$, $\texttt{Sensor}_2$: `Sensor`; $\texttt{Config}_1$: `Config`). In Figure 3(a), both `Sensor` components are located on the same host, whereas in Figure 3(b) each `Sensor` component is located on a different host, which is representative of a component being redeployed to a different host at runtime in a dynamic enterprise DRE system.

Although the runtime composition of the system is different between Figure 3(a) and Figure 3(b), the dataflow model for both deployment (a) and (b) is

the same. As shown in Figure 4, the dataflow model defined by `Sensor (S)` → `Config (C)`. Moreover, the dataflow model in Figure 4 applies to this example

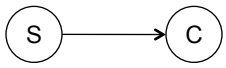

Figure 4: Dataflow Model For Example Deployments in Figure 3

irrespective of the number of `Sensor` and `Config` components (*e.g.*, adding/removing components in at runtime) and their deployment locations (*e.g.*, moving components to different hosts at runtime) in a dynamic enterprise DRE system. As long as either component's behavior remains unchanged, the `Sensor` component continues to send events to the `Config` component. This property is also true for more complex dataflow models, such as the one illustrated in Figure 6 in Section 3.5. The remainder of this section discusses how UNITE integrates this characteristic of dataflow models with system execution traces to validate enterprise DRE system properties.

3.2. Specification and Extraction of Metrics from Text-based System Execution Traces

Analyzing system execution traces is a common technique for validating DRE system behavior [23], particularly functional properties. Execution traces also contain data that can be used to validate QoS properties. For example, Listing 1 shows an example system execution traces produced by a DRE system that must authenticate clients before allowing them access to its resources.

```
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 System Execution Trace Produced by an Enterprise DRE System

Each line in the system execution trace shown in Listing 1 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 execution trace contains data to analyzing the system that produced it, the trace 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—called a *log format*—that will not change throughout the lifetime of system execution. Instead, certain values (or variables) in each log format (*e.g.*, 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.

Using this definition of a log format LF, Equation 1 determines the set of variables in a given log format LF

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

where $vars(LF)$ is a function that extracts the set of variables V for a given log format LF.

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 from Section 2.

Table 1 lists the different placeholder types currently supported by UNITE. UNITE caches the variables and converts the high-level construct into a regular

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

expression. The regular expression is used during the analysis process (see Section 3.4) to identify messages that have candidate data for variables V in log format LF.

Listing 2 exemplifies high-level constructs for two log entries from Listing 1.

```
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

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). Although there are 5 tags in Listing 2, only two tags 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, as described in Section 3.3.

3.3. Specification of Dataflow Models for Evaluating QoS properties

Section 3.2 discussed how UNITE 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 simplest case, a single log format can be used to analyze QoS properties. For example, if developers want to know how many events a component received per second 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 an enterprise DRE 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. What is needed instead is the capability to generate independent log messages and specify how to associate the messages with each other to preserve data integrity. This capability can be accomplished using a dataflow model.

In the context of evaluating QoS properties, we formally define a dataflow model as $DM = (LF, CR, f)$ 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 [24]. In contrast, UNITE uses log format variables to resolve causality because it alleviates dependencies on (1) using a globally unique identifier (*e.g.*, a unique id generated at the beginning of a system execution trace and propagated through the system) and (2) requiring knowledge of system composition to associate metrics (or data). Users must thus only 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).[1]

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 dataflow models in UNITE. Users of UNITE define dataflow models by selecting what log formats should be used to extract data

[1]UNITE does not permit circular relations since it requires human feedback to determine where the relation chain between log formats begins and ends.

from system execution traces. If a dataflow model 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. Users must also define an evaluation function based on the variables in selected log formats.

For example, when QED developer want to calculate duration of the login operation they create a dataflow model 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:

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

Section 3.4 shows how UNITE processes dataflow models of enterprise DRE systems using the specified QoS evaluation function f.

3.4. Evaluation of Dataflow Models

Section 3.2 discussed how UNITE uses log formats to identify messages that contains data of interest and Section 3.3 discussed how it uses log formats and casual relations to specify dataflow models to evaluate QoS properties. The final phase of the UNITE process evaluates the dataflow model, *i.e.*, the evaluation function f. Before explaining the algorithm UNITE uses to process a dataflow model's evaluation function, we first summarize the four types of causal relations that can occur in a component-based DRE systems and can affect the algorithm used to evaluate a dataflow model, as shown in Figure 5. The first type (a)

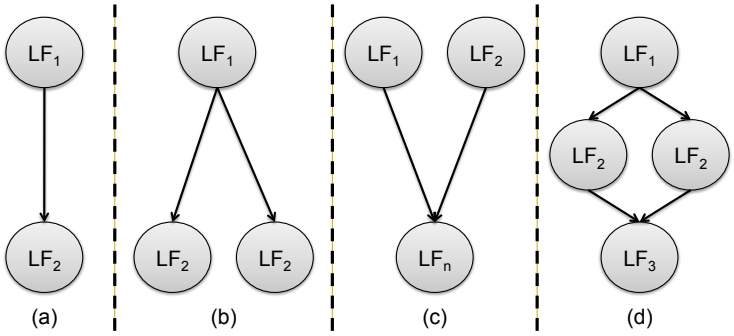

Figure 5: Four Types of Causal Relations in Enterprise DRE Systems

is one-to-one relation, which is the easiest 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)–(c), and is the most complex relation to resolve between multiple log formats.

If we assume that each entry in a message log contains its origin, *e.g.*, hostname, then we can use dynamic programming algorithm [25] and relational database theory to reconstruct the data table of values for a dataflow model's variables in $O(n)^2$ where n is the number of log formats defined in the dataflow model DM. Algorithm 1 shows how UNITE evaluates a dataflow model DM

Algorithm 1 General Algorithm Evaluating a Dataflow Model in UNITE

```
1:  procedure EVALUATE(DM, LM)
2:      DM: dataflow model to evaluate
3:      LM: set of log messages with data
4:      G ← directed_graph(DM)
5:      LF' ← topological_sort(G)
6:      DS ← variable_table(DM)
7:      LM' ← sort LM ascending by (origin, time)
8:
9:      for all LF_i ∈ LF' do
10:         K ← C_i from CR_{i,j}
11:
12:         for all LM_i ∈ LM' do
13:             if matches(LF_i, LM_i) then
14:                 V' ← values of variables in LM_i
15:
16:                 if K ≠ ∅ then
17:                     R ← 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' ← purge incomplete rows from DS
27:     return f(DS') where f is evaluation function for DM
28: end procedure
```

by first creating a directed graph G where log formats LF are nodes and the casual relations $CR_{i,j}$ are edges. UNITE then topologically sorts the directed graph so it knows the order to process each log format. This step is necessary because when causal relation types (a)–(d) are in the dataflow model specifica-

[2]The complexity of this algorithm does not take into account the runtime complexity for selecting the log message from the system execution trace that match a given log format since that depends heavily on how the data is stored in the database and the implementation of the `SELECT` and `REGEXP` function.

tion, processing the log formats in reverse order of occurrence reduces algorithm complexity for constructing data set DS. Moreover, it ensures UNITE has 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, UNITE constructs a data set DS, which is a table that has a column for each variable in the log formats of the dataflow model.[3] UNITE constructs the dataset by first sorting the log messages by origin and time to ensure it has the correct message sequence for each origin. This step is also necessary if users want to see data trends over the system lifetime before aggregating the results, as described in Section 4.2.

UNITE then matches each log format in LF' against each log message in LM'. If there is a match, then UNITE extracts 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, then UNITE locates 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, UNITE appends the values from the log message to the end of the data set. Finally, UNITE purges all the incomplete rows from the data set and evaluate the data set using the user-defined evaluation function for the dataflow model.

Handling duplicate data entries. For long running DRE 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 of a dataflow model to identify variable portions of a message (see Section 3.2). 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. QoS properties, however, are a single scalar 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 aggregation functions include 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 5, the values in the data set for each sender (*i.e.*, LF_2) could be considered a group and analyzed independently.

UNITE requires users to specify an aggregation function as part of the evaluation equation f for a dataflow model because it is known *a priori* whether a QoS evaluation will produce a dataset with multiple values. We formally define a dataflow model with groupings $DM' = (DM, \Gamma)$ as:

[3] An optimization to reduce the size of the data set would be to only insert columns for variables that appear in either the casual relations or evaluation function for the dataflow model.

Table 2: Example Data Set Produced from Evaluating Dataflow Model

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

- A dataflow model DM for evaluating a QoS property; and
- A set $\Gamma \subseteq vars(DM)$ of variables from the log formats in the dataflow model.

Evaluating dataflow models in UNITE. UNITE implements Algorithm 1 using the SQLite relational database (`sqlite.org`). To construct the variable table, the data values for the first log format are first inserted directly into the table since it has no causal relations. For the remaining log formats, the causal relation(s) is transformed into a SQL `UPDATE` query, which allows UNITE 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 dataflow model in Section 3.3. After the variable data table is constructed, the evaluation function and groupings for the dataflow model are used to create the final SQL query that evaluates it, thereby addressing Limitation 2 from Section 2.

```
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 dataflow model—would be 16.33 msec.

3.5. Managing the Complexity of Dataflow Models

Sections 3.2 through 3.4 discussed how UNITE uses dataflow models to evaluate enterprise DRE system QoS properties. Although dataflow models enable UNITE to evaluate QoS properties independent of system implementation and composition, as dataflow models increase in size (*i.e.*, number of log formats and relations between log formats) it becomes harder for DRE system developers to manage their complexity. This challenge arises since dataflow models are similar to *finite state machines* (*i.e.*, the log formats are the states and the relations are the transitions between states), which incur state-space explosion problems [26].

To ensure efficient and effective application of dataflow models to evaluate enterprise DRE system QoS properties, UNITE leverages a model-driven engineering [27] technique called *domain-specific modeling languages (DSMLs)* [28, 29]. DSMLs capture both the semantics and constraints of a target domain while providing intuitive abstractions for modeling and addressing concerns within

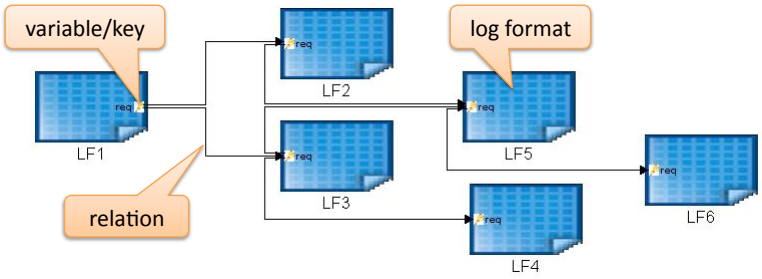

Figure 6: Example Dataflow Model in GME

the target domain. In the context of dataflow models, UNITE's DSMLs provide graphical representations that reduce the following complexities:

- **Visualizing dataflow.** To construct a dataflow model, it is essential to understand dataflow throughout the system, as shown in Figure 5. An invalidate understanding of dataflow can result in an invalid specification of a dataflow model. By using DSMLs, DRE system developers can construct dataflow models as graphs, which helps visualize dataflow and ensure valid construction of such models, especially as such models increase in size and complexity.

- **Enforcing valid relations.** The relations in a dataflow model enable evaluation of QoS properties independent of system composition. Invalid specification of a relation, however, can result in invalid evaluation of a dataflow model. For example, DRE system developers and testers may relate a variable between two different log formats that are of a different type (*e.g.*, one is of type `INT` and the other is of type `STRING`), but have the same variable name (*e.g.*, `id`). The use of DSMLs helps enforce constraints and ensure these invalid relations are not possible in constructed models.

DSMLs in UNITE. UNITE implements several DSMLs using an MDE tool called the *Graphical Modeling Environment (GME)* [30]. GME allows system and software engineers, such as DRE system developers and testers, to author DSMLs for a target domain, such as dataflow modeling. Users then construct models using the specified DSML and use model interpreters to generate concrete artifacts from constructed models, such as a configuration file that specifies how UNITE evaluates a dataflow graph.

Figure 6 shows an example dataflow model for UNITE in GME. Each rectangular object in this figure (*i.e.*, `LF1` and `LF2`) represents a log format in the dataflow model that contains variables for extracting metrics of interest from system execution traces (see Section 3.2). The lines between two log formats represent a relation between variables in either log format. When DRE system developers and testers create a relation between two different variables, the DSML validates the connection (*i.e.*, ensures the variable types are equal).

Likewise, DRE system developers and testers can execute the GME constraint checker to validate system constraints, such as validating that the dataflow model is acyclic (see Section 3.3).

After constructing a dataflow model using UNITE's DSML, DRE system developers and testers use model interpreters to auto-generate configuration files that dictate how to evaluate enterprise DRE system QoS properties. The configuration file is a dense XML-based file that would be tedious and error-prone to create manually. UNITE's DSML graphic representation and constraint checking reduces management complexity. Its auto-generation capabilities also improve specification correctness, thereby addressing Limitation 3 from Section 2.

4. Evaluating UNITE in the QED Project Case Study

This section analyzes results of experiments we conducted to evaluate how UNITE's ability to validate enterprise DRE system QoS properties, and challenges of applying CUTS to the QED project as described in Section 2.

4.1. Experiment Setup

As mentioned in Section 2, the QED project is in its early phases of development. Although it 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 [19] and UNITE to perform early integration testing. All tests were run in the ISISlab testbed (`www.isislab.vanderbilt.edu`), which is powered by Emulab software [31][4]. 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, QED developers first constructed several scenarios using CUTS' modeling languages [32]. Each scenario was designed so that the system was dynamic (*i.e.*, deployed/removed components at runtime) and all components communicate with each other using a single server in the GIG (similar to Figure 2 in Section 2). The first scenario's aim was 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* that tests 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 privileges.

The QED multi-stage workflow has the six types of components shown in Figure 7. 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

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

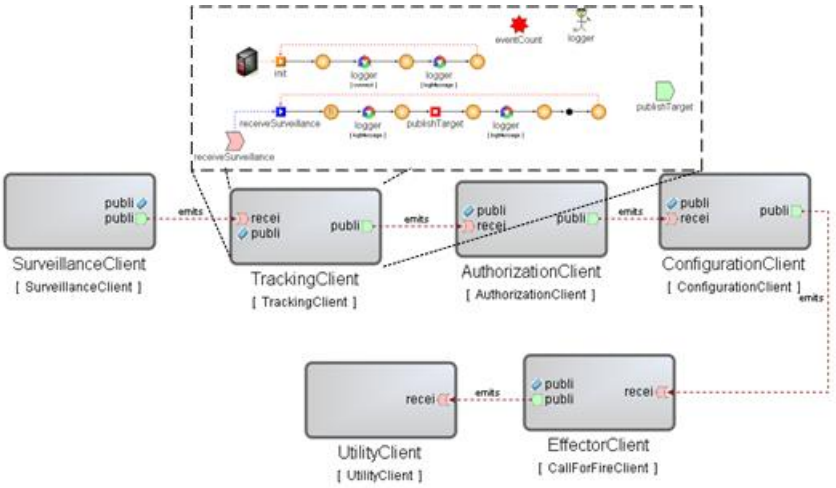

Figure 7: CUTS Model of the Multi-Stage Workflow Test Scenario

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 [32]) that dictates its actions during a test. Moreover, each behavior model contains actions for logging key data needed to evaluate QoS properties, similar to Listing 1 in Section 3.2.

Listing 4 shows an example message from the QED 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.

QED developers next used UNITE to construct log formats (see Section 3.2) for identifying log messages during system execution run that contain metrics of interest. These log formats were also used to define dataflow models that evaluate QoS properties described in Section 3.3. In particular, QED developers were interested in validation the following QoS properties using UNITE:

- **Multiple publishers.** At any point in time, the GIG will have many components publishing and receiving events simultaneously. QED devel-

opers 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. To improve the QoS of the GIG middleware, however, QED developers 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, QED developers cannot ensure it will generate log messages that can be used to calculate how it takes the server to process an event. Instead, QED developers must rely on messages generated from DRE system application components whenever they publish/send events.

For events that propagate through the system, QED developers use Equation 5 to calculate how much time the event spends in the server assuming event transmission is instantaneous, *i.e.*, negligible.

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

This equation also shows how QED developers 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}.

4.2. Experiment Results

This section discusses the results for experiments of the scenarios introduced in Section 4.1. These results are based primarily on the QoS properties of concern discussed in Section 4.1.

4.2.1. Analyzing Multiple Publisher Results

Table 3 presents the results for tests that measure average end-to-end response time for an event when each publisher publishes at 75 Hz. As expected,

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

the response time for each importance value was similar. When we 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—but accounted for adding/removing components at runtime (*i.e.*, the dynamic nature of the system). The total number of log messages generated during the course of the system execution was 993,493.

UNITE also allows QED developers and testers to view the data trend for the dataflow models QoS evaluation of this scenario to get a more detailed understanding of performance. Figure 8 shows how the response time of the event increases over the lifetime of the experiment. Although QED developers and

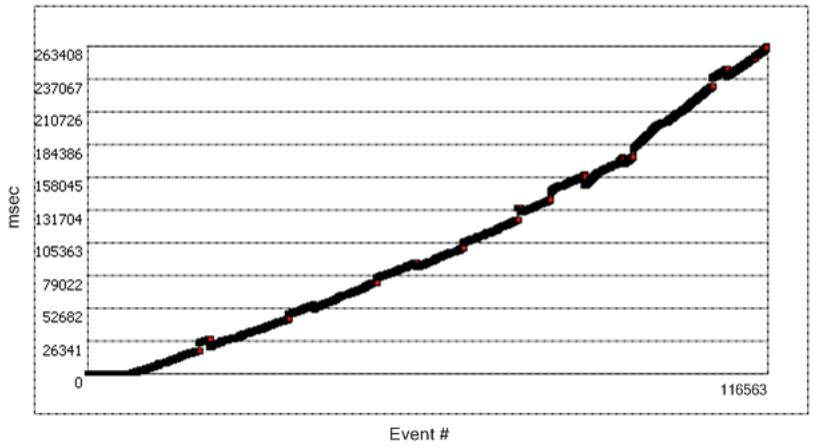

Figure 8: Data Trend Graph of Average End-to-End Response Time for Multiple Publishers Sending Events at 75 Hz

testers hypothesized that this test configuration produced too much workload, UNITE's data trend and visualization capabilities clearly quantified the extent to which the GIG middleware was over utilized. Without UNITE, moreover, QED developers and testers would not have had a tool that could (1) adapt to the dynamic nature of their DRE system and (2) provide detailed analysis thatis hard to obtain using conventional methods.

4.2.2. Analyzing Maximum Sustainable Publish Rate Results

QED developers used the multi-stage workflow to describe a complex dynamic scenario that tested the limits of the GIG middleware without forcing it to queue events incrementally. Figure 9 graphs the data trend for the test, which is calculated by specifying Equation 5 as the evaluation for the test, and was produced by UNITE after analyzing (*i.e.*, identifying and extracting metrics from) 193,464 log messages. The test also consisted of ten different log formats and nine different causal relations, which were of types (a) and (b), as discussed in Section 3.4.

Figure 9 shows the sustainable publish rate of the mutle-stage workflow in ISISlab. This figure shows how the Java just-in-time (JIT) compiler and other Java features cause the QED middleware to temporarily increase the individual message end-to-end response. By the end of the test (which is not shown in

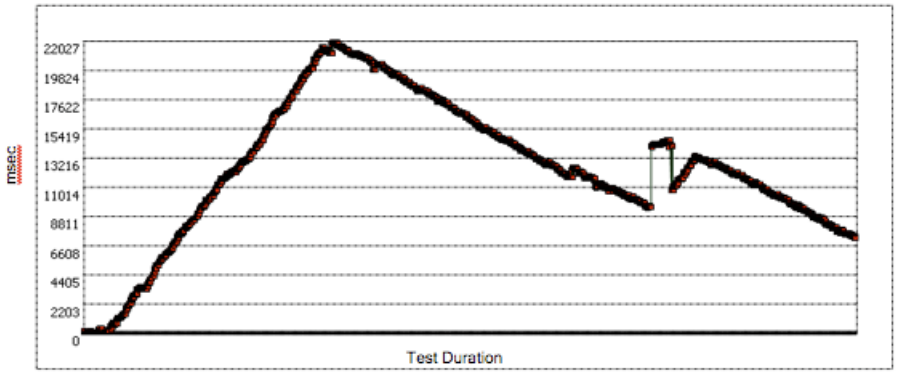

Figure 9: Data Trend of the System Placed in Near-Optimal Publish Rate

Figure 9), 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 hypothesis of the maximum sustainable publish rate in ISISlab was confirmed. Second, the dynamic analytical capabilities of UNITE that produced Figure 9 helped developers pinpoint which features of the GIG middleware might cause performance bottlenecks; how QED could address such problems; and what new tests are needed to showcase QED's improvements to the GIG middleware. By providing QED testers comprehensive testing and analysis features, UNITE helped guide the QED development team's next phase of testing and integration of feature sets.

In terms of applying UNITE to the multi-stage workflow, QED developers did not have to propagate a unique id for each event through each component to validate end-to-end response time. If UNITE required a global unique identifier to associate data metrics, as in conventional approaches, QED developers would then need to ensure that all components propagated the unique identifier to accurately validate QoS properties, which would be hard for dynamic enterprise DRE systems. Moreover, if QED developers added new components to the multi-stage workflow, each component would need a global unique identifier. Accommodating global unique identifiers would complicate the logging specification, system structure (to propagate the unique id), and require more resources (such as CPU, network bandwidth, and memory).

4.3. Evaluating the Scalability of UNITE

As enterprise DRE systems (such as the GIG/QED middleware and their applications) increase in size and complexity, UNITE's corresponding dataflow models also increase in size and complexity. Moreover, the amount of data that must be processed by a dataflow model to evaluate QoS properties also increases in size. Algorithm 1 presents UNITE's algorithm dataflow graph that QED developers use to evaluate QoS properties of the GIG middleware. The

run-time complexity of this algorithm depends mainly on the number of log formats in the dataflow graph. Its runtime complexity is also dependent on the number of variables that appear in a relation because this affects the run-time complexity of correlating two separate log formats.

Table 4 presents the results of evaluating the scalability of UNITE with respect to the number of log formats and relation variables in a dataflow model. Each result in the figure was generated by executing a test that generated a

Table 4: Execution Time (Secs) for Evaluating Dataflow Models

	# of Log Formats (L)				
# of Relation Vars (R)	1	3	5	10	20
1	1.542	2.754	3.529	4.424	8.645
3	2.31	7.872	11.363	16.119	27.519
5	2.53	10.789	19.706	44.236	109.795
10	4.935	15.85	27.903	**169.642**	163.457
20	7.34	26.239	50.967	122.601	317.792

system execution trace where each log format contained 20,000 messages, and each single message had 1 correlation with another log format. The results show that as either the number of log formats or relation variables increase, the overall execution time of the QoS evaluation also increases. In the case of 10 log formats and 10 relation variables (*i.e.*, test 10L-10R), however, the execution time does not follow this trend.

To explain why the data point in Table 4 does not follow the trend, we next examine the size of the dataset used to generate these initial execution times. Table 5 shows the size of the data set for each test in Table 4. As shown in

Table 5: Dataset Size (MB) for System Execution Trace

	# of Log Formats (L)				
# of Relation Vars (R)	1	3	5	10	20
1	1.5	2.0	2.5	4.0	6.9
3	4.5	6.2	7.8	12.1	20.8
5	7.5	10.3	12.9	20.3	34.7
10	15.1	20.8	25.9	**81.9**	69.461
20	30.7	42.08	52.287	81.734	142.210

Table 5, the size of the dataset for the test does affect the overall execution time. In the test case 10L-10R the generated dataset size for the test was greater, even though there are either fewer relations than test 10L-20R or test 20L-10R.

Our analysis of these test results indicate that as the number of log formats and relation variables increase, the overall evaluation time increases. Moreover, the evaluation time is also directly dependent on the size of the dataset, irrespective of the number of log formats and relation variables. QED developers and testers thus realized that they should focus more on reducing how much

data is collected to ensure evaluation times remain low and UNITE's analytical process remains lightweight.

5. Related Work

This section compares our work on UNITE with related work validate system properties with system execution traces, uses of dataflow modeling, and enterprise DRE system QoS analysis.

System execution traces. System execution traces capture system state and metrics. With this in mind, Chang et al. [3] have investigated techniques for automating the functional validation of test using execution traces. Likewise, Moe et al. [2] discuss techniques for understanding and detecting functional anomalies in distributed systems by reconstructing and analyzing system execution traces. Irrespective of how system execution traces are used, their key advantage to validating functional concerns is platform-, architecture-, and language-independence. This therefore helps increase the quality of the overall solution [33] so that it is applicable across different application domains. The main difference between UNITE and the existing research above is that UNITE extends their effort to QoS properties, such as end-to-end response time, scalability, and throughput. UNITE's analytical process and algorithm is also applicable to dynamic DRE systems (*i.e.*, ones that change the structure and composition throughout the system's execution lifetime).

Dataflow modeling. Dataflow models, also known as dataflow diagrams, have been used extensively in software design and specification [7, 34] digital signal processing [35] and business processing modeling [8]. For example, Vazquez invested techniques for automatically deriving dataflow models from formal specifications of software systems. Likewise, Russell et al. investigate the feasibility of using UML activity diagrams within business logic process models, which include dataflow modeling. In all cases, dataflow modeling was utilized because it provides a means for representing system functionality without being bound to the system's overall composition. This is because the dataflow models, in theory, remain constant unless the system's specification changes.

UNITE enhances existing research on dataflow model usage by applying them on validating QoS properties via system execution traces. Moreover, dataflow models are used to (1) preserve data integrity (or causality) when reconstructing the dataset that contains all the data points of interest in a system execution trace and (2) analyze extracted data (or metrics) independent of system structure, composition, and complexity.

Early enterprise distributed system testing. Coelho et al. [36] and Yamany et. al [37] describe techniques for 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 QoS concerns of DRE systems during the early stages of development. Moreover, Coelho et al. test a single multi-agent isolation, whereas UNITE tests and evaluates systemic properties (*i.e.*, many components working together), as well as components in isolation.

Qu et. al [38] present a tool named *DisUnit* that extends JUnit [39] to enable unit testing of component-based distributed systems. Although DisUnit supports testing of distributed systems, it assumes that metrics used to evaluate a QoS property are produced by a single component. As a result, DisUnit cannot evaluate distributed system QoS properties 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.

Enterprise DRE system QoS analysis. Mania et. al [5] 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 embedded systems, the reconstructed behavior and analysis may be incorrect if clocks on different hosts drift, as is often the case in enterprise DRE systems. UNITE improves 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.

Mos et al. [4] present a similar 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 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. [6] present a technique for performing end-to-end event tracing in component-based distributed systems by injecting 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 their technique by relaxing the need for a global unique identifier to associate data for analysis. Moreover, in large- or ultra-large-scale enterprise DRE systems, it can be hard to ensure that unique identifiers are propagated throughout components created by third parties. Since UNITE does not rely on global identifiers it can reconstruct system behavior for analysis even if components developed by third parties do not generate any events or log messages.

6. Concluding Remarks

System execution traces can used to validate both enterprise DRE system functional and QoS properties. Conventional ways of applying system execution traces to validate QoS properties, however, are overly coupled to system implementation and composition details. This paper described and evaluated the *Understanding Non-functional Intentions via Testing and Experimentation*

(UNITE) tool, which uses dataflow models to validate QoS properties of enterprise DRE systems independently from system implementation and composition details. UNITE's techniques for validating QoS properties are lightweight and scalable, *i.e.*, they depend primarily on the amount of data processed, as opposed to the size and complexity of the dataflow model that reflects system size, composition, and complexity.

The following is a summary of lessons learned based on results and experience developing and applying UNITE to a representative enterprise DRE system:

- **Dataflow modeling increases the level of abstraction for evaluating QoS properties.** Instead of requiring complete knowledge of system composition and implementation, UNITE's dataflow models provide an platform-, architecture-, and technology-independent technique for evaluating the performance of DRE system QoS properties.

- **Use of DSMLs simplifies data-flow modeling.** DSMLs provide an intuitive visual interface for constructing dataflow models and auto-generating configuration files needed by UNITE. They also reduce the occurrence of accidental errors via their constraint checking mechanisms, such as invaliding relations, reducing of redundant data, and removing unnecessary data.

- **Creating dataflow models is a time-consuming and error-prone task.** Although UNITE's DSML was designed to reduce complexities associated with defining and managing dataflow models, it is tedious and error-prone to ensure their specification will extract the correct metrics due to the disconnect between the log messages used to generate execution traces and log formats that extract metrics these log messages in system execution traces. Our future work will therefore investigate techniques for auto-generating dataflow models from system execution traces.

- **Parallelization is needed to help decrease evaluation time.** Our empirical results showed that dataset size had more effect on evaluation time than the number of log formats or relation variables in a dataflow model. Future work therefore will investigate techniques for parallelizing evaluation of dataflow models so evaluation time is not dependent on the size of the dataset (or system execution traces).

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

[1] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, C. D. Gill, QoS-enabled Middleware, in: Q. Mahmoud (Ed.), Middleware for Communications, Wiley and Sons, New York, 2004, pp. 131–162.

[2] J. Moe, D. A. Carr, Understanding Distributed Systems via Execution Trace Data, in: International Workshop on Program Comprehension, 2001.

[3] F. Chang, J. Ren, Validating system properties exhibited in execution traces, in: ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, New York, NY, USA, 2007, pp. 517–520. doi:http://doi.acm.org/10.1145/1321631.1321723.

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

[5] D. Mania, J. Murphy, J. McManis, Developing Performance Models from Nonintrusive Monitoring Traces, IT&T.
URL `citeseer.ist.psu.edu/541104.html`

[6] T. Parsons, Adrian, J. Murphy, Non-Intrusive End-to-End Runtime Path Tracing for J2EE Systems, IEEE Proceedings Software 153 (2006) 149–161.

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

[8] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, P. Wohed, On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling, in: Proceedings of the 3rd Asia-Pacific Conference on Conceptual modelling, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2006, pp. 95–104.

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

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

[11] S. E. Institute, Ultra-Large-Scale Systems: Software Challenge of the Future, Tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA (June 2006).

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

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

[14] M. El-Gendy, A. Bose, K. Shin, Evolution of the internet qos and support for soft real-time applications, Proceedings of the IEEE 91 (7) (July 2003) 1086–1104. doi:10.1109/JPROC.2003.814615.

[15] Rittel, H. and Webber, M., Dilemmas in a General Theory of Planning, Policy Sciences (1973) 155–169.

[16] J. Mann, The role of project escalation in explaining runaway information systems development projects: A field study, Ph.D. thesis, Georgia State University, Atlanta, GA (1996).

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

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

[19] J. H. Hill, J. Slaby, S. Baker, 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, 2006.

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

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

[22] J. Hill, D. C. Schmidt, J. Slaby, 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, 2008.

[23] N. Joukov, T. Wong, 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, 2005, pp. 25–25.

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

[25] R. Bellman, Sequential Machines, Ambiguity, and Dynamic Programming, Journal of the ACM 7 (1) (1960) 24–28.

[26] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231–274.
URL citeseer.ist.psu.edu/article/harel87statecharts.html

[27] D. C. Schmidt, Model-Driven Engineering, IEEE Computer 39 (2) (2006) 25–31.

[28] J. Sztipanovits, G. Karsai, Model-Integrated Computing, IEEE Computer 30 (4) (1997) 110–112.

[29] J. Gray, J. Tolvanen, S. Kelly, A. Gokhale, S. Neema, J. Sprinkle, Domain-Specific Modeling, in: CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.), CRC Press, 2007, pp. 7.1–7.20.

[30] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, Computer 34 (11) (2001) 44–51. doi:http://dx.doi.org/10.1109/2.963443.

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

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

[33] B. W. Boehm, J. R. Brown, M. Lipow, Quantitative Evaluation of Software Quality, in: Proceedings of the 2^{nd} International Conference on Software Engineering, 1976, pp. 592–605.

[34] A. A. A. Jilani, A. Nadeem, T. hoon Kim, E. suk Cho, Formal representations of the data flow diagram: A survey, Advanced Software Engineering and Its Applications 0 (2008) 153–158. doi:http://doi.ieeecomputersociety.org/10.1109/ASEA.2008.34.

[35] E. A. Lee, T. M. Parks, Dataflow Process Networks (2002) 59–85.

[36] R. Coelho, U. Kulesza, A. von Staa, 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, 2006, pp. 83–90.

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

[38] R. Qu, S. Hirano, T. Ohkawa, T. Kubota, R. Nicolescu, Distributed Unit Testing, Tech. Rep. CITR-TR-191, University of Auckland (2006).

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