SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2009; **00**:1–22 (DOI: 10.1002/000)

Using Dataflow Models to Evaluate Enterprise Distributed Real-time and Embedded System Quality-of-Service

James H. Hill[1,†,*], Pooja Varshneya[2], Hamilton A. Turner[2],
James R. Edmondson[2], and Douglas C. Schmidt[2]

[1] *Department of Computer and Information Science, Indiana University/Purdue University at Indianapolis, 723 W. Michigan Street, SL 280, Indianapolis, IN 46202-5132*
[2] *Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37203*

SUMMARY

The effort required to evaluate enterprise distribute real-time and embedded (DRE) system quality-of-service (QoS) attributes (such as response-time, latency, and scalability) depends heavily on system complexity and size. As these systems increase in complexity and size, therefore, DRE system developers and testers need improved methods and tools that facilitate QoS evaluation. This article describes a method and tool called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)* that evaluates enterprise DRE system QoS attributes using dataflow models to capture how data move through an enterprise DRE system. Empirical results show that although UNITE's evaluation times depend on the size of the dataflow model, they depend even more on the size of the dataset processed by the dataflow model. Copyright © 2009 John Wiley & Sons, Ltd.

Received X August 2009

KEY WORDS: enterprise DRE systems; dataflow models; quality-of-service evaluation; early system integration testing; system execution traces; relational database theory

1. Introduction

Challenges of enterprise distributed real-time and embedded (DRE) system testing. Enterprise DRE systems (*e.g.*, shipboard computing environments, urban traffic management systems, and air traffic control systems) are a class of systems that must satisfy functional (*e.g.*, operational

---

*Correspondence to: Department of Computer and Information Science, Indiana University/Purdue University at Indianapolis, 723 W. Michigan Street, SL 280, Indianapolis, IN 46202-5132
†E-mail: hillj@cs.iupui.edu

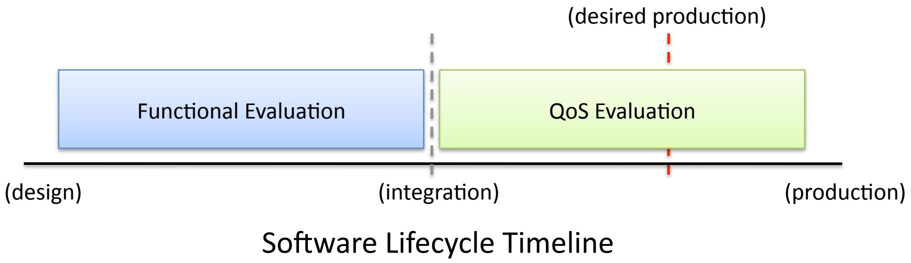


Figure 1. Differences of functional and QoS evaluation in enterprise DRE systems.

capabilities) *and* quality-of-service (QoS) requirements (*e.g.*, end-to-end response time, throughput, and scalability) [1]. Functional attributes of enterprise DRE systems are often evaluated throughout the software lifecycle using common methods and tools, such as test-driven development [2, 3], unit testing [4, 5], and continuous integration services [3, 6, 7]. In contrast, QoS attribute evaluation often occurs late in the software lifecycle (*e.g.*, during the system integration phase) since accurately evaluating these attributes historically required a complete system fielded in its target environment (which includes the hardware/software resources) [8].

Deferring QoS attribute evaluation in DRE systems can severely impact cost, schedule, and quality. For example, deadlines due to failure to meet QoS requirements can result in costly and length software lifecycles at the expense of its stakeholders [9]. This problem is exacerbated by the disconnect between functional and QoS attribute evaluation, where the former typically occurs continuously throughout the software lifecycle and the latter typically does not [10]. As shown in Figure 1, DRE system developers and testers are often forced to delay the release of production software because QoS requirements were not met due the disconnect between functional and QoS attribute evaluation.

System execution modeling (SEM) [11–14] tools help bridge the gap between understanding how functional and QoS attributes affect each other. SEM tools provide DRE developers and testers with artifacts to emulate the constructed models on the target environment understand QoS attribute performance earlier in the software lifecycle, *i.e.*, before complete system integration. Moreover, SEM tools support incrementally replacement of the emulated "faux" portions of the system with real implementations as development progresses. This capability enables DRE system developers and testers to conduct *continuous system integration testing* [15], which evaluates both functional *and* QoS attributes continuously throughout the software lifecycle.

Although SEM tools are a promising approach, conventional SEM tools have significant limitations that arise when emulated portions of an enterprise DRE system are replaced with real implementations. For example, developers and tester can no longer rely on built-in analytical capabilities (such as software probes that measure end-to-end response-time) of SEM tools that are embedded in emulated portions of the DRE system. Instead, they must manually implement these strategies that collect and analyze metrics when evaluating QoS attributes.

Conventional techniques for manually collecting and analyzing metrics, however, are tightly coupled to (1) *system implementation* [16, 17], *i.e.*, what technologies are used to implement the system, and (2)

DOI: 10.1002/stvr

system composition [17, 18], *i.e.*, where components are located and what components communicate with each other. These constraints can limit analytical capabilities as a DRE system evolves throughout its software lifecycle. New techniques are therefore needed to improve the analytical capabilities of enterprise DRE system QoS attributes so it is not tightly coupled to system complexity and system implementation.

Solution approach → Higher-level QoS evaluation. To evaluate QoS attributes independently of system composition and implementation, these evaluations should be performed at a higher-level of abstraction, similar to how domain-specific modeling languages [19, 20] help create enterprise DRE systems independent of system implementation [21]. A promising technique is to use *dataflow models* [22], which describe how data flows through an information system, because a dataflow remains constant irrespective of system composition and implementation. In the context of enterprise DRE systems, dataflow models describe 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 be used to extract metrics collected while executing the DRE system in its target environment and evaluate its QoS attributes.

Although dataflow models enable evaluation of QoS attributes independent of system composition and implementation, it is hard to construct these models and evaluating them efficiently and effectively with conventional techniques. To address this problem, we describe a method and tool called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)* that utilizes dataflow models to evaluate enterprise DRE system QoS attributes. UNITE analyzes dataflow models using relational database theory [23] techniques where metrics used to evaluate a QoS attribute are associated with each other via their relations in the dataflow model. The constructed metric's table is then evaluated by applying an SQL expression based on a user-defined function.

DRE system developers and testers can use UNITE to evaluate QoS attributes continuously throughout the software lifecycle via the following steps shown in Figure 1 and summarized below:

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 a dataflow model that is used to extract metrics of interest based for QoS evaluation; and
4. Define a QoS evaluation equation to analyze a dataflow model and evaluate a QoS attribute, such as end-to-end response time, latency, and scalability.

Our experience applying UNITE to a representative enterprise DRE system shows it is an effective technique for evaluating QoS attributes through the software lifecycle (including early and later phases) without depending on system composition and implementation details. Moreover, its evaluation evaluation capabilities is more dependent on the amount of data processed as opposed to the size of the dataflow model. This therefore provides a more scalable solution for evaluating enterprise DRE system QoS attributes because the dataflow model is lesser of a factor on evaluation capabilities.

Article organization. The remainder of this article is organized as follows: Section 2 summarizes a representative DRE system case study to motivate the need for UNITE; Section 3 describes UNITE's structure and functionality; Section 4 presents the results of experiments that measure the benefits of

Prepared using stvrauth.cls
DOI: 10.1002/stvr

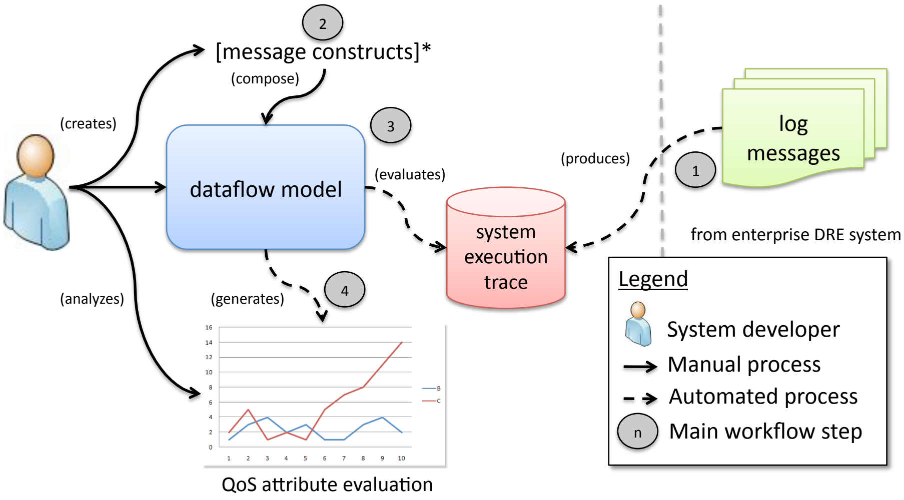


Figure 2. Overview of UNITE's workflow.

applying UNITE to our case study; Section 5 compares UNITE with related work that evaluates DRE system QoS attributes and data; and Section 6 presents concluding remarks.

2. Case Study: the QED Project

The Global Information Grid (GIG) middleware [24] is a enterprise DRE system from the class of ultra-large-scale (ULS) systems [25]. 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) [26] 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 3 shows QED in the context of the GIG. At the heart of the QED middleware is a Java 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 [27] and differentiated service queues [28] to provide QoS assurance to GIG applications.

DOI: 10.1002/stvr

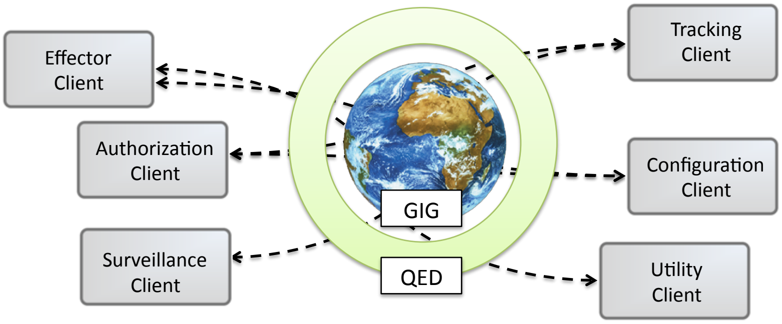


Figure 3. Conceptual Model of QED in the Context of the GIG.

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 enterprise DRE system development: the *serialized-phasing development problem* [29]. In serialized-phasing 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 attributes, however, are typically not discovered until the final stages of development, *e.g.*, at system integration time, which is too late in the software lifecycle [9, 30].

To overcome the serialized-phasing problem, QED developers are using SEM tools to automatically execute performance regression tests against the QED and evaluate QoS attributes continuously throughout its development. In particular, QED is using the *Component Workload Emulator (CoWorkEr) Utilization Test Suite* (CUTS) [12], which is a platform-independent SEM tool for enterprise DRE systems. 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. DRE system developers and testers then execute the test system on their target architecture, and CUTS collects performance metrics, which can be used to evaluate QoS attributes. This process is then repeated continuously throughout the software lifecycle in increase confidence levels in QoS assurance.

Our prior work [15] showed how integrating CUTS-based SEM tools 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 showed how simple log messages can capture metrics of interest to evaluate QoS attributes continuously throughout the software lifecycle. 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

DOI: 10.1002/stvr

time. It was therefore hard to identify, locate, and extract data for metrics of interest, especially if QoS evaluation functions needed data that CUTS did not know *a priori*, such as metrics extracted from a real component that replaces an emulate component, 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 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. Sections 3.1 and 3.2 describe how UNITE addresses this limitation by using log formats and dataflow models to evaluate QoS attributes within system execution traces.

- **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 evaluation traditionally yields a scalar that determines 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, and implied analysis was tightly couple to system implementation and system complexity.

  QED testers need a flexible technique for collecting metrics that can be used in user-defined functions to evaluate various system-wide QoS attributes, 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 3.3 describes how UNITE addresses this limitation by using relational database theory techniques to evaluate dataflow models.
- **Limitation 3: Inability to manage complexity of QoS attribute evaluation specification.** As enterprise DRE systems increase in size and complexity, the challenges associated with limitations 1 and 2 described above will also increase in complexity. For example, as DRE system implementations mature more components are often added and the amount of data generated for QoS attribute evaluation will increase. Likewise, the specification of a QoS attribute 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 to QED testers can focus more on QoS attribute evaluation as opposed to specification of QoS attribute evaluation. Section 3.4 describes how UNITE addresses this limitation by using domain-specific modeling languages.

Due to the limitations described above, it was hard for QED developers to use the initial version of CUTS to conduct QoS evaluation continuously throughout the software lifecycle without being tightly coupled to both system implementation and system complexity. Moreover, this problem extends beyond the QED project and applies to other enterprise DRE systems that want to perform QoS evaluation continuously throughout the software lifecycle. The remainder of this article shows how

UNITE addresses these limitations by improving CUTS to evaluate QoS attributes without being dependent on system implementation and composition.

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

This section presents the underlying theory of UNITE and describes how it uses dataflow models to facilitate system implementation- and composition-independent QoS evaluation of enterprise DRE system QoS attributes.

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, regardless of whether the system is distributed [31]. 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 system execution trace produced by an enterprise DRE 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—called 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

DOI: 10.1002/stvr

Table I. 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

message that may change over the course of the systems lifetime, thereby addressing Limitation 1 from Section 2. Table I 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.

```
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 Dataflow Models for Evaluating QoS Attributes

Section 3.1 discussed 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 simplest case, a single log format can be used to analyze QoS attributes. 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 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. Instead, what is needed 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 attributes, 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. UNITE, in contrast, 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). 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, however, since 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 I), *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. In UNITE, users define dataflow models by selecting what log formats should be used to extract data from message logs. 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. Last, users define an evaluation function based on the variables in selected log formats.

For example, if a QED developer wanted to calculate duration of the login operation, then 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:

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

The following section discusses how we process dataflow models of enterprise DRE systems using the specified QoS evaluation function f.

3.3. Evaluation of Dataflow Models

Section 3.1 discussed how UNITE uses log formats to identify messages that contains data of interest and Section 3.2 discussed it uses log formats and casual relations to specify dataflow models to evaluate QoS attributes. The final phase of the UNITE process involves evaluating the dataflow model, *i.e.*, the evaluation function f. Before we explain the algorithm UNITE uses to process a dataflow model's

DOI: 10.1002/stvr

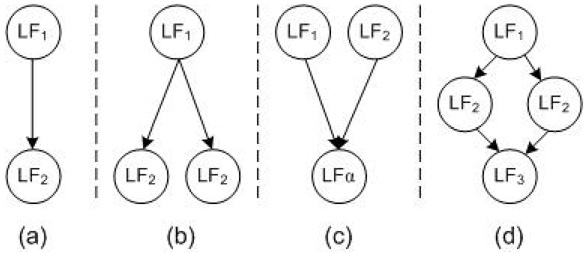


Figure 4. Four types of causal relations that can occur in enterprise DRE systems.

evaluation function, it is necessary to first understand different types of causal relations that can occur in an enterprise DRE system.

Four types of causal relations that can occur in a component-based distributed system and can affect the algorithm used to evaluate a dataflow model are shown in Figure 4. 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)–(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 and relational database theory to reconstruct the data table of values for a dataflow model's variables. As shown in Algorithm 1, UNITE evaluates a dataflow model DM 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 specification, 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.[†] 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 is also 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.

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

---

[†] 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.

Algorithm 1 General algorithm evaluating a dataflow model in UNITE.

```
procedure EVALUATE($DM, LM$)
    $DM$: dataflow model to evaluate
    $LM$: set of log messages with data
    $G \leftarrow directed\_graph(DM)$
    $LF' \leftarrow topological\_sort(G)$
    $DS \leftarrow variable\_table(DM)$
    $LM' \leftarrow$ sort $LM$ ascending by (origin, time)

    for all $LF_i \in LF'$ do
        $K \leftarrow C_i$ from $CR_{i,j}$

        for all $LM_i \in LM'$ do
            if $matches(LF_i, LM_i)$ then
                $V' \leftarrow$ values of variables in $LM_i$

                if $K \neq \emptyset$ then
                    $R \leftarrow findrows(DS, K, V')$
                    $update(R, V')$
                else
                    $append(DS, V')$
                end if
            end if
        end for
    end for

    $DS' \leftarrow$ purge incomplete rows from $DS$
    return $f(DS')$ where $f$ is evaluation function for $DM$
end procedure
```

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 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 3.1). 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. QoS attributes, however, are a single scalar value, and not multiple values. To address this concern, we use the following techniques:

DOI: 10.1002/stvr

Table II. 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

- **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 4, 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 dataflow model because it is known *a priori* if a QoS evaluation will produce a dataset with multiple values. We formally define a dataflow model with groupings $DM' = (DM, \Gamma)$ as:

- A dataflow model DM for evaluating a QoS attribute; 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 II shows the variable table constructed by UNITE for the example dataflow model in Section 3.2. 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 II. The final result of this example—and the dataflow model—would be 16.33 msec.

3.4. Managing the Complexity of Dataflow Models

Sections 3.1 through 3.3 discussed how UNITE uses dataflow models to evaluate enterprise DRE system QoS attributes. Although dataflow models enable UNITE to evaluate QoS attributes independent of system implementation and composition, as dataflow models increase in size (*i.e.*,

DOI: 10.1002/stvr

number of log formats and relations between log formats) is 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 [32].

To ensure efficient and effective application of dataflow models towards evaluating enterprise DRE system evaluating QoS attributes, UNITE leverages a model-driven engineering [33] technique called *domain-specific modeling languages (DSMLs)* [19, 20]. DSMLs capture both the semantics and constraints of a target domain while providing intuitive abstractions for modeling and addressing concerns within the target domain. In the context of dataflow models, 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 4 in Section 3.3. 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 attribute 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 tests 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`). By using DSMLs, it is possible to enforce constraints that will ensure such relations are not possible in constructed models.

DSMLs in UNITE. UNITE implements several DSMLs using an MDE tool called the *Graphical Modeling Environment (GME)* [34]. GME allows system and software engineers, such as DRE system developers and testers, to author DSMLs for a target domain, such as dataflow modeling. End-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 show evaluate a dataflow graph.

Figure 3.4 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 3.1). 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 DSMLs 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 systemic constraints, such as validating that the dataflow model is acyclic (see Section 3.2).

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 attributes. 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.

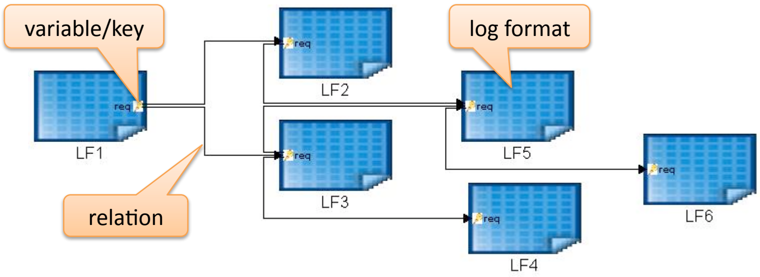


Figure 5. Example dataflow model in GME.

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 and the limitations with CUTS described in Section 2.

4.1. Experiment Setup

As mentioned in Section 2, the QED project is in its first year 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 [12] 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 [35][‡]. 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 [36]. Each scenario was designed so that all components communicate with each other using a single server in the GIG (similar to Figure 3 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* 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 six types of components, as shown in Figure 6. 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.

[‡] 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.

DOI: 10.1002/stvr

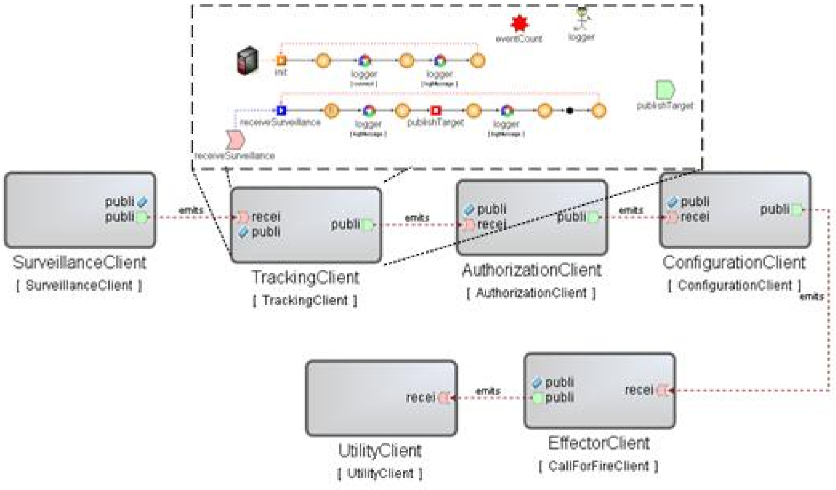


Figure 6. CUTS model of the multi-stage workflow test scenario

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 [36]) that dictates its actions during a test. Moreover, each behavior model contains actions for logging key data needed to evaluate QoS attributes, 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.

QED developers next used UNITE to construct log formats (see Section 3.1) for identifying log messages during a test run that contain metrics of interest. These log formats were also used to define dataflow models that evaluate QoS attributes described in Section 3.2. In particular, QED developers were interested in evaluating the following QoS attributes using dataflow models in UNITE:

- **Multiple publishers.** At any point in time, the GIG will have many components publishing and receiving events simultaneously. QED developers therefore need to evaluate the response

DOI: 10.1002/stvr

[hptb]
Table III. 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

time of events under such operating conditions. Moreover, QED needs to ensure QoS when the infrastructure servers must manage many events. In order 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 distributed application components whenever it publishes/sends an event.

For an event that propagates 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}.

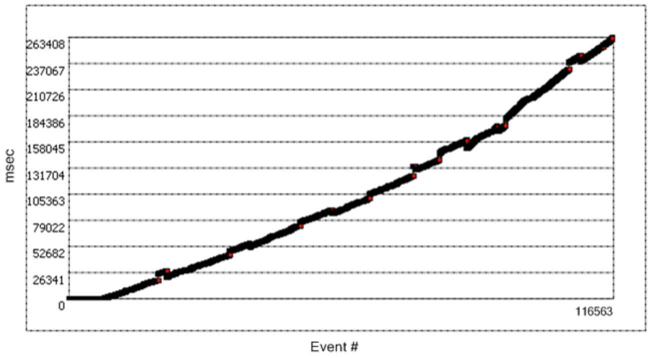
4.2. Experiment Results

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

4.2.1. Analyzing Multiple Publisher Results

Table III presents the results for tests 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 tested this scenario using UNITE, the test results presented in Table III 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.

UNITE also allows QED developer 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 7 shows how



[hptb]

Figure 7. Data trend graph of average end-to-end response time for multiple publishers sending events at 75 Hz

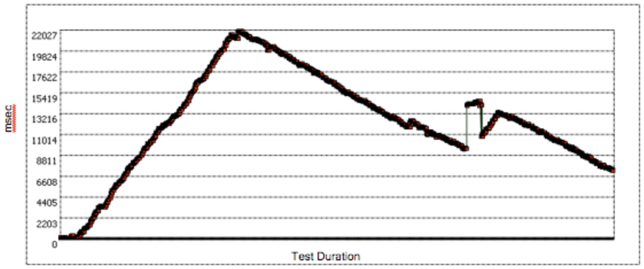


Figure 8. Data trend of the system placed in near optimal publish rate

the response time of the event increases over the lifetime of the experiment. We knew beforehand that the this configuration for the test produced too much workload. UNITE's data trend and visualization capabilities, however, helped make it clear the extent to which the GIG middleware was being over utilized.

4.2.2. Analyzing Maximum Sustainable Publish Rate Results

QED developers 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 8 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 3.3.

DOI: 10.1002/stvr

[htpb]
Table IV. Execution time (secs) for evaluating dataflow models.

	# of Log Formats (L)				
# of Relation Variables (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

Figure 8 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 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 8 helped developers speculate on what features of the GIG middleware might cause performance bottlenecks, how QED could address such problems, and what new 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.

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 will also increase in size and complexity. Moreover,b the amount of data that must be processed by a dataflow model to evaluate QoS attributes will also increase is size. Algorithm 1 presented UNITE's algorithm dataflow graph that QED developers use to evaluate QoS attributes 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 IV 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 system execution trace where each log format contained 20,000 messages, and each single message had 1 correlation with another log format. The results also show that as either the number of log formats or relation variables increase, the overall execution time of the QoS evaluation 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.

Table V. Comparison of dataset size (MB) vs. evaluation time (sec) in UNITE.

	# of Log Formats (L)				
# of Relation Variables (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

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

4.4. Evaluating the Impact of UNITE on the Experiments

UNITE enabled QED developers to quickly construct dataflow models to evaluate QoS attributes of the GIG middleware (Section 4.1). In the context of the QED multi-stage workflow scenario, UNITE provided two insights to QED developers. First, their theory of maximum publish rate in ISISlab was confirmed (Section 4.2.1). 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 test are need to showcase QED's improvements to the GIG middleware (Section 4.2.2).

In addition, UNITE's analytical capabilities are not bounded system complexity and composition. As long as the correct log formats and their causal relations is specified, UNITE can evaluate QoS attributes. QED developers also did not need to specify a global unique identify to associate data with its correct execution trace. If UNITE required a global unique identifier to associate data metrics, then QED developers would have to ensure that all components propagated the identifier. Moreover, if QED developers added new components to the multi-stage workflow, each component would have to be aware of the global unique identifier, which can inherently complicate the logging specification.

Finally, the scalability results showed QED developers that the complexity and size of the dataflow model has lesser impact on evaluation time of dataflow models (Section 4.3). Instead, they should focus more on reducing how much data is collected to ensure evaluation times remain low. By providing comprehensive testing and analysis capabilities, UNITE helped guide QED developers through their next phase of testing and integration of feature sets by reducing the complexity of evaluating enterprise DRE system QoS attributes.

DOI: 10.1002/stvr

5. Related Work

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

Early enterprise distributed system testing. Coelho et al. [37] and Yamany et. al [38] 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 non-functional concerns of a distributed system during the early stages of development. Moreover, Coelho et al. test a single multi-agent isolation, whereas UNITE focuses on testing and evaluating systemic properties (*i.e.*, many components working together). UNITE can also be used to test and evaluate a component in isolation, if necessary.

Qu et. al [39] present a tool named *DisUnit* that extends JUnit [4] 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 attribute are produced by a single component. As a result, DisUnit cannot be used to evaluate a QoS attribute 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.

Enterprise DRE system QoS analysis. Mania et. al [18] 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. [16] 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. [17] 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 large- or ultra-large-scale enterprise DRE systems, it can be hard to ensure unique identifiers are propagated throughout components created by third parties. Since UNITE does not rely on the global identifier, it can reconstruct system behavior for analysis even if the component's not developed in-house do not produce any events (or log messages).

6. Concluding Remarks

QoS attributes of enterprise distributed real-time and embedded (DRE) systems have traditionally been tested during final system integration, which can severely impact cost, schedule, and quality. The earlier QoS attributes are tested in the actual target environment, therefore, the greater the chances of locating and remedying performance-related problems in a timely and cost-effective manner [40, 41]. This paper describes and evaluates a technique and tool called *Understanding Non-functional Intentions via Testing and Experimentation (UNITE)* for evaluating QoS attributes of enterprise DRE systems throughout the software lifecycle. UNITE enables DRE system developers to evaluate QoS attributes irrespective of system implementation and composition. Moreover, UNITE can be used to evaluate QoS attributes without *a priori* knowledge of the equations required to evaluate the desired attributes.

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

- **Dataflow modeling increases the level of abstraction for evaluating QoS attributes.** Instead of requiring knowledge of system composition and implementation, dataflow models provided an platform-, architecture-, and technology-independent technique for evaluating QoS attributes.
- **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.** Results showed that the size of the dataset 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 in open-source format for download at `www.dre.vanderbilt.edu/CUTS`.

REFERENCES

1. Wang N, Schmidt DC, Gokhale A, Rodrigues C, Natarajan B, Loyall JP, Schantz RE, Gill CD. QoS-enabled Middleware. *Middleware for Communications*, Mahmoud Q (ed.). Wiley and Sons: New York, 2004; 131–162.
2. Janzen D, Saiedian H. Test-Driven Development: Concepts, Taxonomy, and Future Direction. *IEEE Computer* 2005; **38**(9):43–50.
3. Bowyer J, Hughes J. Assessing Undergraduate Experience of Continuous Integration and Test-driven Development. *Proceeding of the 28th International Conference on Software Engineering (ICSE'06)*, 2006; 691–694.
4. Massol V, Husted T. *JUnit in Action*. Manning Publications Co.: Greenwich, CT, USA, 2003.
5. Hunt A, Thomas D. *Pragmatic Unit Testing in C# with NUnit*. The Pragmatic Programmers: Raleigh, NC, USA, 2004.
6. Holck J, Jorgenson N. Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects. *Australasian Journal of Information Systems* 2003–2004; :40–53.
7. Fowler M. Continuous Integration. www.martinfowler.com/articles/ continuousIntegration.html May 2006.
8. Denaro G, Polini A, Emmerich W. Early Performance Testing of Distributed Software Applications. *ACM SIGSOFT Software Engineering Notes* January 2004; **29**(1):94–103.

DOI: 10.1002/stvr

9. Snow A, Keil M. The Challenges of Accurate Project Status Reporting. *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, 2001.
10. Ho CW, Johnson MJ, Williams L, Maximilien EM. On agile performance requirements specification and testing. *Proceedings of Agile 2006*, 2006.
11. Smith C, Williams L. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional: Boston, MA, USA, 2001.
12. Hill JH, Slaby J, Baker S, Schmidt DC. Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, 2006.
13. Box D, Shukla D. WinFX Workflow: Simplify Development with the Declarative Model of Windows Workflow Foundation. *MSDN Magazine* 2006; **21**:54–62.
14. Dutoo M, Lautenbacher F. Java Workflow Tooling (JWT) Creation Review. www.eclipse.org/proposals/jwt/JWT%20Creation%20Review%2020070117.pdf 2007.
15. Hill J, Schmidt DC, Slaby J, Porter A. CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments. *Proceeedings of 15th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*, Belfast, Northern Ireland, 2008.
16. Mos A, Murphy J. Performance Monitoring of Java Component-Oriented Distributed Applications. *IEEE 9th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2001; 9–12.
17. Parsons T, Adrian, Murphy J. Non-Intrusive End-to-End Runtime Path Tracing for J2EE Systems. *IEEE Proceedings Software* August 2006; **153**:149–161.
18. Mania D, Murphy J, McManis J. Developing Performance Models from Nonintrusive Monitoring Traces. *IT&T* 2002; URL `citeseer.ist.psu.edu/541104.html`.
19. Sztipanovits J, Karsai G. Model-Integrated Computing. *IEEE Computer* Apr 1997; **30**(4):110–112.
20. Gray J, Tolvanen J, Kelly S, Gokhale A, Neema S, Sprinkle J. Domain-Specific Modeling. *CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.)*. CRC Press, 2007; 7.1–7.20.
21. Balasubramanian K. Model-Driven Engineering of Component-based Distributed, Real-time and Embedded Systems. PhD Thesis, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville Sep 2007.
22. Downs E, Clare P, Coe I. *Structured systems analysis and design method: application and context*. Prentice Hall International (UK) Ltd.: Hertfordshire, UK, UK, 1988.
23. Atzeni P, Antonellis VD. *Relational Database Theory*. Benjamin-Cummings Publishing Co., Inc.: Redwood City, CA, USA, 1993.
24. Global Information Grid. The National Security Agency, www.nsa.gov/ia/industry/ gig.cfm?MenuID=10.3.2.2.
25. Institute SE. Ultra-Large-Scale Systems: Software Challenge of the Future. *Technical Report*, Carnegie Mellon University, Pittsburgh, PA, USA Jun 2006.
26. Loyall J, Carvalho M, Schmidt D, Gillen M, III AM, Bunch L, Edmondson J, Corman D. QoS Enabled Dissemination of Managed Information Objects in a Publish-Subscribe-Query Information Broker. *Defense Transformation and Net-Centric Systems*, 2009.
27. Tortonesi M, Stefanelli C, Suri N, Arguedas M, Breedy M. Mockets: A Novel Message-Oriented Communications Middleware for the Wireless Internet. *International Conference on Wireless Information Networks and Systems (WINSYS 2006)*, 2006.
28. El-Gendy M, Bose A, Shin K. Evolution of the internet qos and support for soft real-time applications. *Proceedings of the IEEE* July 2003; **91**(7):1086–1104, doi:10.1109/JPROC.2003.814615.
29. Rittel, H and Webber, M. Dilemmas in a General Theory of Planning. *Policy Sciences* 1973; :155–169.
30. Mann J. The role of project escalation in explaining runaway information systems development projects: A field study. PhD Thesis, Georgia State University, Atlanta, GA 1996.
31. Joukov N, Wong T, Zadok E. Accurate and Efficient Replaying of File System Traces. *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, 2005; 25–25.
32. Harel D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* June 1987; **8**(3):231–274. URL `citeseer.ist.psu.edu/article/harel87statecharts.html`.
33. Schmidt DC. Model-Driven Engineering. *IEEE Computer* 2006; **39**(2):25–31.
34. Lédeczi Á, Bakay Á, Maróti M, Völgyesi P, Nordstrom G, Sprinkle J, Karsai G. Composing Domain-Specific Design Environments. *Computer* 2001; **34**(11):44–51, doi:http://dx.doi.org/10.1109/2.963443.
35. Ricci R, Alfred C, Lepreau J. A Solver for the Network Testbed Mapping Problem. *SIGCOMM Computer Communications Review* Apr 2003; **33**(2):30–44.
36. Hill JH, Gokhale A. Model-driven Engineering for Early QoS Validation of Component-based Software Systems. *Journal of Software (JSW)* Sep 2007; **2**(3):9–18.
37. Coelho R, Kulesza U, von Staa A, Lucena C. Unit Testing in Multi-agent Systems using Mock Agents and Aspects. *International Workshop on Software Engineering for Large-scale Multi-agent Systems*, 2006; 83–90.

38. Yamany HFE, Capretz MAM, Capretz LF. A Multi-Agent Framework for Testing Distributed Systems. *30th Annual International Computer Software and Applications Conference*, 2006; 151–156.
39. Qu R, Hirano S, Ohkawa T, Kubota T, Nicolescu R. Distributed Unit Testing. *Technical Report CITR-TR-191*, University of Auckland 2006.
40. Weyuker EJ. Testing Component-based Software: A Cautionary Tale. *Software, IEEE* Sep/Oct 1998; **15**(5):54–59.
41. Wu Y, Chen MH, Offutt J. UML-Based Integration Testing for Component-Based Software. *Proceedings of the Second International Conference on COTS-Based Software Systems*, Springer-Verlag, 2003; 251–260.