Data Mining System Execution Traces to Validate Distributed System Quality-of-Service Properties

Dr. James H. Hill
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
723 W. Michigan Ave., SL 280D
Indianapolis, IN 46202

ABSTRACT

System Execution Modeling (SEM) tools enable distributed system testers to validate Quality-of-Service (QoS) properties, such as end-to-end response time, throughput, and scalability, during early phases of the software lifecycle. Analytical capabilities of QoS properties, however, are traditionally bounded by a SEM tool's capabilities. This chapter discusses how to mine system execution traces, which are a collection of log messages describing events and states of a distributed system throughout its execution lifetime, generated by distributed systems so that the validation of QoS properties is not dependent on a SEM tool's capabilities. The author uses a real-life case study to illustrate how data mining system execution traces can assist in discovering potential performance bottlenecks using system execution traces.

INTRODUCTION

Challenges of enterprise distributed system development

Enterprise distributed systems, such as mission avionic systems, traffic management systems, and shipboard computing environments, are transitioning to next-generation middleware, such as service-oriented middleware (Pezzini & Natis, 2007) and component-based software engineering (Heineman & Councill, 2001). Although next-generation middleware is improving enterprise distributed system functional properties (i.e., its operational scenarios), Quality-of-Service (QoS) properties (e.g., end-to-end response time, throughput, and scalability) are not validated until late in the software lifecycle, i.e., during system integration time. This is due in part to the *serialized-phasing development* problem (Rittel & Webber, 1973).

As illustrated in Figure 1, in serialized-phasing development, the infrastructure- and application-level system entities, such as components that encapsulate common services, are developed during different phases of the software lifecycle. Software design decisions that affect QoS properties, however, are typically not discovered until final stages of development, e.g., at system integration time, which is too late in the software lifecycle to resolve performance bottlenecks in an efficient and cost effective manner (Mann, 1996; Snow & Keil, 2001; Woodside, Franks, & Petriu, 2007).

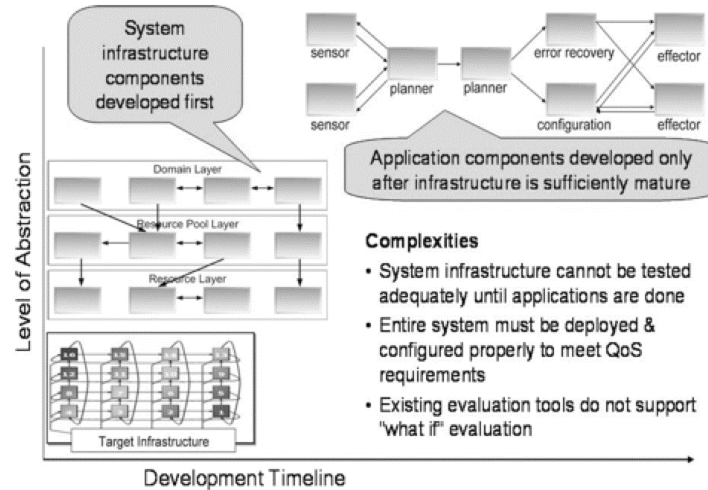

Figure 1. Overview of serialized-phased development in distributed systems

System Execution Modeling (SEM) tools (Smith & Williams, 2001), which are a form of model-driven engineering (Schmidt, 2006), assist distributed system developers in overcoming the serialized-phasing development problem shown in Figure 1. SEM tools use domain-specific modeling languages (Ledeczi, Maroti, Karsai & Nordstrom, 1999) to capture both platform-independent attributes (such as structural and behavioral concerns of the system) and platform-specific attributes (such as the target architecture of the system) as high-level models. Model interpreters then transform constructed models into source code for the target architecture. This enables distributed system testers to validate QoS properties continuously throughout the software lifecycle while the "real" system is still under development. Likewise, as development of the real system is complete, distributed system testers can incrementally replace *faux* portions of the system with its real counterpart to produce more realistic QoS validation results.

Although SEM tools enable distributed system developers and testers to validate distributed system QoS properties during early phases of the software lifecycle, QoS validation capabilities are typically bounded to a SEM tool's analytical capabilities. In order to validate QoS properties unknown to a SEM tool, distributed system testers have the following options:

- **Use handcrafted solutions.** This option typically occurs outside of the SEM. Moreover, this option is traditionally not applicable across different application domains because it is an *ad hoc* solution, e.g., handcrafting a solution to validate event response-time based on priority in a proprietary system;

- **Leverage model transformations to convert models to a different SEM tool.** This option implies the source and target SEM tool supports the same modeling features and semantics. If the target SEM tools has different modeling features and semantics, then distributed system testers have discrepancies in QoS validation results (Denton, Jones, Srinivasan, Owens, & Buskens, 2008); or

- **Wait for updates to the SEM tool.** This is the best option for distributed system testers because it ensures consistency of QoS validation results when compared to the previous two options. In many cases, however, this option may not occur in a timely manner so that distributed system testers can leverage the updates in their QoS validation exercises. Distributed system testers therefore have to revert to either of the first two options until such updates are available, which can result in the problems previously discussed.

Consequently, relying solely on built-in validation capabilities of SEM tools can hinder distributed system testers to thoroughly validate enterprise distributed system QoS properties continuously throughout the software lifecycle. Distributed system testers therefore need improved techniques that will enhance QoS analytical capabilities irrespective of the SEM tools existing capabilities.

Solution approach → QoS validation using system execution traces. To address problems associated with limited analytical capabilities of a SEM tool when validating QoS properties, there is a need for methodologies that extend conventional SEM tool methodologies and simplify the following exercises, as illustrated in Figure 2:

1. **Capturing QoS property metrics** without the SEM tool having *a priori* knowledge of what metrics (or data) is required to analyze different QoS properties. This step can be accomplished using system execution traces (Chang & Ren, 2007), which are a collection of log messages generated during the execution lifetime of a distributed system in its target environment. The log messages in the system execution trace are lightweight and flexible enough to adapt the many different QoS metrics that formulate throughout the software lifecycle and across different application domains;

2. **Identifying QoS property metrics** without requiring *a priori* knowledge of what data (or metrics) is being collected (i.e., the ability to learn at run-time). This step can be accomplished using *log formats*, which are expressions that identity the static and variable portions of log messages of interest within system execution traces generated in Step 1. The log formats are then used to mine system execution traces and extract metrics of interest for QoS validation; and

3. **Evaluating QoS properties** without *a priori* knowledge of how to analyze extracted QoS metrics. This step can be accomplished using dataflow models (Downs, Clare, & Coe, 1988) that enable distributed system testers auto-reconstruct end-to-end system execution traces for QoS validation. Distributed system testers then specify a domain-specific (i.e., user-defined) equation for validating QoS properties using metrics data minded in Step 2.

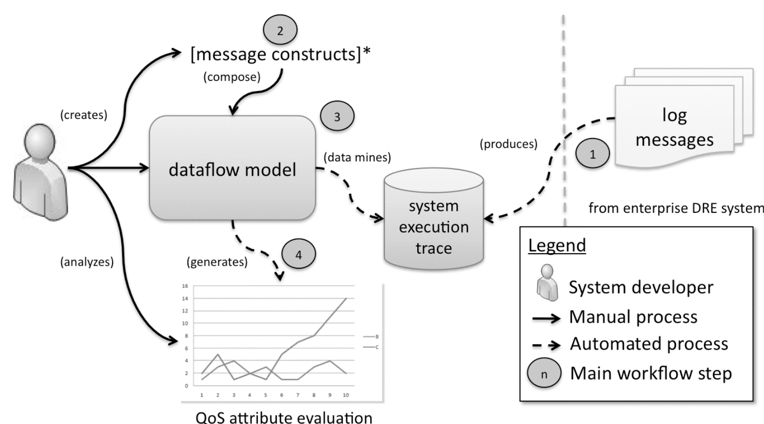

Figure 2. Overview of using dataflow models to mine system execution traces and validate QoS properties

Using dataflow models to mine system execution traces enable distributed system testers to validate QoS properties independent of the SEM tool of choice. Likewise, as enterprise distributed systems continue increasing in size (e.g., number of lines of source code, and number of hardware/software resources) and complexity (e.g., envisioned operational scenarios), dataflow models can adapt without modification. This is because dataflow models operate at a higher level of abstraction than system composition (i.e., how components communicate with each other) and system complexity (i.e., the operational nature of the system in its target environment). Likewise, domain-specific analytics associated with dataflow models need not change.

This chapter illustrates the following concepts for using dataflow models to mine system execution traces and validate enterprise distributed system QoS properties:

- How to use high-level constructs to specify QoS metrics that are to be extracted from system execution traces;
- How to represent high-level constructs as dataflow models to ensure correct auto-reconstruction of end-to-end system execution traces; and
- How to use dataflow models to mine system execution traces and validate enterprise distributed system QoS properties using domain-specific analytical equations.

Distributed system testers therefore can focus more on using SEM tools to discover QoS bottlenecks specific to their application domain, and are ensured they will be able to perform such activities irrespective of a SEM tool's analytical capabilities.

BACKGROUND

Before beginning the discussion on using dataflow models to mine system execution traces and validate QoS properties, first let us understand existing techniques used to validate enterprise distributed system QoS properties. This section therefore summarizes current uses of system execution traces, data mining, and dataflow modeling, and conventional techniques for validating distributed systems QoS properties.

System execution traces

System execution traces can capture both metrics and the state of a system. Keeping this in mind, Chang and Ren (2007) has investigated techniques for automating the functional validation of test using execution traces. Likewise, Moe and Carr (2001) discuss techniques for understanding and detecting functional anomalies in distributed systems by reconstructing and analyzing (or data mining) system execution traces. Irrespective of how system execution traces are used, their key advantage to validating functional concerns is platform, architecture, and language independence. Therefore this helps to increase the quality of the overall solution (Boehm, Brown & Lipow, 1976) so that it is applicable across different application domains.

Data mining

Data mining system execution traces to locate faults is a well-studied research topic. For example, Denmat, Ducasse and Ridoux (2005) reinterpreted Jones, Harrold and Stasko (2002) problem to visualizing faulty statements (i.e., statements that could be potential sources of a bug) in execution traces as a data mining problem. In doing so, Denmat et al. (2005) were able to uncover

limitations in Jones et al. solution approach and address some of the limitations. Likewise, Lo et al. (2007) contribute to the domain of *specification mining* (Ammons, Bodik, & Larus, 2002 ; Lo & Khoo, 2006), which is the process of inferring specifications from execution traces. In their work, Lo et al. (2007) developed several data mining algorithms that can reconstruct scenarios from execution traces for visualization as UML2 Sequence Diagrams. Finally, Lo et al. (2008) developed algorithms for data mining execution traces to discover temporal rules about program execution, which can help understand system behavior and improve program verification.

Although data mining system execution traces has been heavily researched (e.g., locating faults and understanding program behavior), existing research focuses primarily on functional properties of the system. Data mining system execution traces to validate QoS properties, such as end-to-end response time, throughput, and scalability, has not been studied in much detail – especially in the context of distributed systems. This chapter therefore provides contributions on enabling data mining of system execution traces for distributed systems and validation of QoS properties.

Distributed system analysis

Mania, Murphy and McManis (2002) discusses a technique for developing performance models and analyzing component-based distributed system using execution traces. The contents of traces are generated by system events. When analyzing the systems performance, however, (Mania et al., 2002) 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. Similarly, (Mos & Murphy, 2001) presents 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.

Parsons et al. (2006) presents 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 (i.e., to preserve data integrity). In large- or ultra-large-scale enterprise distributed systems, however, it can be hard to ensure unique identifiers are propagated throughout components created by third parties.

Dataflow modeling

Dataflow models, also known as dataflow diagrams, have been used extensively in software design and specification (Downs, Clare & Coe, 1988; Jilani, Nadeem, Kim & Cho, 2008), digital signal processing (Lee & Parks, 2002), and business processing modeling (Russell, van der Aalst, ter Hofstede & Wohed, 2006). For example, Vazquez (1994) invested techniques for automatically deriving dataflow models from formal specifications of software systems. Likewise, Russell et al. (2006) investigates the feasibility of using UML activity diagrams within business processing process, which include dataflow modeling. In all cases, dataflow modeling was utilized because it provides a means for representing system functionality without being bounded to the systems overall composition. This is because the dataflow models, in theory, remain constant unless the system's specification changes.

To date, little research has investigated the use of dataflow models to mine system execution traces and validate QoS properties. Because system execution traces are dense and rich sources of information, they are good candidates for data mining to validate QoS properties. The main

challenge, however, is extracting information from the system execution traces while (1) preserving data integrity and (2) analyzing extracted data (or metrics) at without *a priori* knowledge its structure and complexity.

As discussed later in this chapter, dataflow modeling is one part of the solution to data mining system execution traces to validate distributed system QoS properties. The dataflow models are used to capture how data is transmitted throughout a distributed system. This information is then used to reconstruct end-to-end system execution traces and preserve data integrity (i.e., ensuring metrics are correlated with their correct chain of events) so the user-defined evaluation function analyzes the metrics correctly.

MOTIVATIONAL CASE STUDY: THE QED PROJECT

The Global Information Grid (GIG) middleware (National Security Agency, 2009) is an enterprise distributed system from the class of Ultra-Large-Scale (ULS) systems (Software Engineering Institute, 2006). 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) project (Loyall, et al., 2009) is a multi-organization collaboration designed to improve GIG middleware so it can meet QoS requirements of users and distributed applications and systems. QED's aim therefore is 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 (Tortonesi, Stefanelli, Suri, Arguedas, & Breedy, 2006) and differentiated service queues (El-Gendy, Bose, & Shin, 2003) to provide QoS assurance to GIG applications.

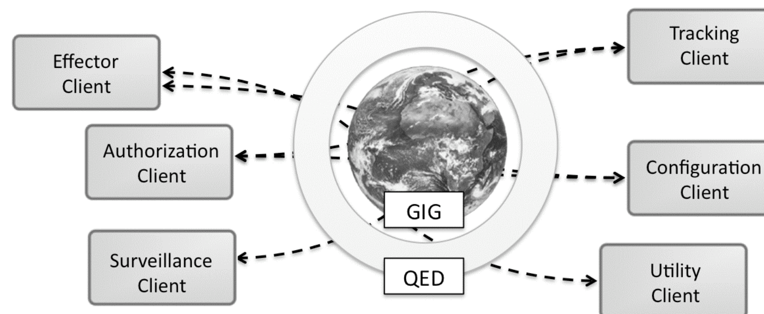

Figure 3. Overview of QED in the context of the GIG middleware

The QED project is in the early phases of its software lifecycle and its development 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 the serialized-phasing development problem as explained in the introduction section.

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) (Hill, Slaby, Baker, & Schmidt, 2006), which is a platform-independent SEM tool for enterprise distributed systems. Distributed system developers and testers use CUTS by modeling the behavior and workload of their enterprise distributed system and generating a test system for their target architecture. Distributed system testers then execute the test system on the target architecture and validate QoS attributes. This process is repeated continuously throughout the software lifecycle to increase confidence levels in QoS assurance.

Previous research showed how integrating CUTS-like SEM tools with continuous integration environments provided a flexible solution for executing and managing distributed system tests continuously throughout the software lifecycle (Hill, Schmidt, Slaby, & Porter, 2008). This work also showed how system execution traces capture metrics of interest for validating QoS properties. Applying results from prior work to the initial prototype of the QED middleware, however, revealed the following limitations of adapting CUTS to the QED project:

- **Limitation 1: Inability to mine metrics of interest unknown *a priori* to SEM tools.** Data mining is the process of discovering relevant information in a data source, such as a system execution trace, that can be used for analysis (e.g., validating QoS properties). In the initial version of CUTS, data mining was limited to metrics that CUTS knew *a priori*, i.e., at compilation 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 therefore need 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 for effective application to enterprise distributed systems. This technique can be realized using log formats, which are high-level abstractions that capture variable and static portions of log messages in system execution traces. The variable portion of each log format is also used to mine and extract metrics of interest from system execution traces, which can later be used in user-defined equations that analyze QoS properties.

- **Limitation 2: Inability to analyze and aggregate extracted data unknown *a priori* to SEM tools.** Data analysis and aggregation is the process of evaluating extracted data based on user-defined equations, and combining multiple results (if applicable) into a single result. This process is necessary since evaluating QoS properties traditionally yields a scalar result, such as evaluating that the response-time of an event is 30.4 msec. 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 data using user-defined functions.

  QED testers needed a flexible technique for analyzing metrics using user-defined functions. Moreover, the technique should preserve data integrity (i.e., ensuring data is associated with its correct execution trace), especially as the system increases in both complexity and size. This technique can be realized using dataflow models and analyzing the dataflow using

relational database theory techniques (Atzeni & Antonellis, 1993).

- **Limitation 3: Inability to manage complexity of dataflow model specification.** As enterprise distributed systems increase in size and complexity, challenges associated with Limitations 1 and 2 described above will also increase in complexity. For example, as distributed 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 by QED testers. This technique can be accomplished using domain-specific modeling languages (Sztipanovits & Karsai, 1997; Gray, Tolvanen, Kelly, Gokhale, Neema & Sprinkle, 2007), which are abstractions that capture the semantics and constraints of a target domain while providing intuitive graphical representations that shield end-users, such as the QED testers, from its complexities.

Due to the limitations described above, it was hard for QED testers to use the initial version of CUTS and validate QoS properties without being dependent on its analytical capabilities. Moreover, this problem extends beyond the QED project and applies to other projects need to validate QoS properties irrespective of a SEM tool's analytical capabilities. The following section discusses how system testers can data mine system execution traces generated by a distributed system to validate QoS properties and address the limitations previously described.

DATA MINING SYSTEM EXECUTION TRACES USING DATA FLOW MODELS

This section describes in detail the technique of using dataflow models to mine system execution traces and validate of distributed system QoS properties independent of the SEM tool's analytical capabilities. This section also describes how the technique is realized in an open-source tool called UNITE, which has been integrated into the CUTS SEM tool. Finally, this section concludes by illustrating how UNITE is applied to the QED project case study.

Specifying QoS Metrics to Mine from System Execution Traces

System execution traces, which are a collection of log messages generated throughout the lifetime of a system executing in its target environment, are essential in understanding the behavior of a system, whether or not the system is distributed (Joukov, Wong, & Zadok, 2005 ; Chang & Ren, 2007). Such artifacts typically contain key data that can be used to analyze the system online and/or offline. For example, Listing 1 shows a simple system execution trace produced by a distributed system that requires password authentication before clients can use its resources.

1 activating LoginComponent
2 *(more log messages)*
3 LoginComponent received 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 *(more log messages)*

8 deactivating the LoginComponent

Listing 1. Example of system execution trace produced by a distributed system

As illustrated in Listing 1, each line in the system execution trace represents an effect in the system. Moreover, each line in Listing 1 captures the state of the system when the log message was produced. For example, line 3 states when LoginComponent received a login request and line 6 captures when LoginComponent granted access to the client.

Although a system execution trace contains key data for analyzing the system that produced it, a system execution trace is traditionally generated in a verbose format that can be understood by humans. This implies that information data captured in a system execution trace can be discarded. Each log message that appears in a system execution trace is also constructed from a well-defined format—called a *log format. Log formats* are high-level constructs that capture both constant and variable portions of individual, yet similar, log messages in a system execution trace. The information captured in the variable portion of a log format represents metrics that is data mined from the system execution trace and is usable in domain-specific analytical equations. This format remains constant throughout the execution lifetime of system, and only certain values (or variables) in each log format, e.g., time or event count, change over time. The challenge therefore is specifying what metrics should be data mined from the system execution trace so that the extracted data is useable in user-defined analytical equations.

Specifying log formats in UNITE. In UNITE, log formats are specified using high-level constructs composed of human readable text and placeholders identified by brackets { } . Table 1 shows the different placeholder types supported by UNITE). The brackets are used to tag variables (or metrics) that are to be data mined from a system execution trace generated by a distributed system. Each placeholder (or bracket) also represents variable change in a log message (such as those presented in Listing 1) over the course of the system's lifetime. This enables UNITE to address Limitation 1 introduced at the latter part of the case study.

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

Table 1. Log format variable types supported by UNITE

UNITE caches the variables and converts the high-level construct into a regular expression, such as a PERL compatible regular expression (www.pcre.org). The regular expression is used during the data mining process to identify log messages that have candidate data for variables in log format.

Log Format:

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

PERL Compatible Regular Expression:

```
LF1: (?<owner>\\S+) received request (?<reqid>-?\\d+) at (?<reqid>-?\\d+)
LF2: granting access at (?<reply>-?\\d+) to request (?<reqid>-?\\d+)
```

Listing 2. Example of log formats for identifying metrics of interest

Listing 2 highlights high-level constructs for two log message entries from Listing 1 and the corresponding PERL compatible regular expression. The first log format (LF1) is used to mine log messages related to receiving client login requests (line 3 in Listing 1). The second log format (LF2) is used to mine log messages related to granting access to a client's login request (line 6 in Listing 1). Overall, there are 5 variables in Listing 2. Only two variables, however, capture metrics of interest: recv in LF1 and reply in LF2. The remaining three variables (i.e*.,* owner, LF1.reqid, and LF2.reqid) are used to determine causality and preserve data integrity.

Preserving Data Integrity during the Data Mining Process

In the log formats that are used for identifying log messages in system execution traces that contain data (or metrics) of interest, each log format contains a set of tags, which are representative of variables that are used to mine metrics for each log format. In the simplest case, a single log format can be used to validate QoS properties. For example, if a distributed system tester wants to know how many events a component received per second, i.e., the arrival rate of events, then the component could cache the necessary data internally and generate a single log message reflecting this metric when the system is shutdown.

Although this approach is feasible, i.e., caching data and generating a single log message, it is not practical in a distributed system because individual data points needed to validate a QoS property can be generated by different components. Moreover, data points can be generated by components deployed on different hosts. What is needed therefore is the capability of generating independent log messages and specifying how to associate independent log messages with each other to preserve data integrity. This capability can be realized using a dataflow model. In the context of data mining system execution traces, a dataflow model is defined as:

- A set of log formats that have variables identifying what data to mine from system execution traces; and
- A set 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.

To understand how dataflow models can be used in this situation in a better way, it is necessary to understand the role of log formats first. As discussed in the previous section, system execution traces capture (ordered) events and the state of a distributed system throughout its execution lifetime. The individual log messages in a system execution trace also can be identified using log formats. Since the log formats are directly related to log messages in a system execution trace, the order of a log format will be the same as the order of occurrence for log messages in a system execution trace. If a dataflow model consists of nodes (i.e*.,* individual data points) and relations (i.e*.,* how data is transmitted between individual nodes), then log formats represent the individual nodes in the dataflow model. More specifically, log format variables represent the individual nodes in a dataflow model since they capture critical pieces of information in system execution traces need to construct it.

After defining the nodes in a dataflow model, the next step is to define the relations between nodes (i.e*.,* the log formats). As a result, this paves the way for defining the causality in the dataflow model. Causal relations are traditionally based on time (Singhal & Shivaratri, 1994),

such as determining that e_1 is causally related to e_2 because $t_1 < t_2$ when using a global clock to timestamp events. This notion of time, however, does not exist in dataflow models used to mine system execution traces. This is because log format variables are used to determine causality since the value of a variable is guaranteed to not change between two causally related log formats, and removes all ambiguity when determining causality of events in dense system execution traces.

Using log format variables to define causality between nodes in the dataflow model also has other several advantages over using time, such as: 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 throughout the system or a global clock) and (2) requiring knowledge of system composition to associate metrics across multiple log messages and to preserve data integrity.

Instead, the only requirement of the dataflow model is that two unique log formats can be associated with each other via their variables, and each log format is in at least one causal relation. This requirement, however, can result in circular relations between log formats. Circular relations therefore are not permitted because it requires human intervention to determine where the relation chain between log formats begins and ends.

Specifying dataflow models in UNITE. In UNITE, distributed system testers specify dataflow models by selecting what log formats are used to mine system execution traces. If more than one log format is needed, then they must specify a causal relation between each log format. When specifying casual relations, distributed system testers select variables from the corresponding log format that represent the cause and effect.

For example, if a QED developer wants to calculate duration of the login operation, then they create a dataflow model using LF1 and LF2 from Listing 2. Next, a causal relation is defined between LF1 and LF2 as:

LF1.reqid = LF2.reqid

Listing 3. Example of a causal relation in a dataflow model

As Listing 3 illustrates, LF1 and LF2 are causally related to each other based on the value of reqid in either log format. Now that it is possible to define a dataflow model in terms of log formats and their causal relations, a discussion of how to evaluate dataflow models using user-defined evaluation functions based on metrics data mined from system execution traces is given in the next section.

Evaluating Dataflow Models using User-defined Equations

The previous sections discussed how log formats are (1) used to identify log messages that contains data of interest, and (2) used to construct dataflow models to mine system execution traces. The main purpose of the dataflow model is to preserve data integrity, which is essential to ensure all data points are associated with the execution trace (or event) that generated it. The final step in the process is therefore to evaluate the dataflow model using a user-defined equation, which represents validating a QoS property that is unknown to a SEM tool. For example, if a distributed system developer wanted to calculate the authentication time based on the system execution trace presented in Listing 1, they would define the following equation:

$$LF2.reply - LF1.recv$$

Listing 4. Example of a user-defined equation for evaluating authentication time

Before discussing how to evaluate a user-defined equation, such as the one presented in Listing 4, using a dataflow model and metrics data mined from system execution traces, it is necessary to first understand the different types of causal relations that can occur in a distributed system.

There are four types of causal relations that can occur in a distributed system and can affect the algorithm used to evaluate a dataflow model. As shown in Figure 4, the first type is (a) one-to-one relation, which is the most trivial type to resolve between multiple log formats. The second type is (b) one-to-many relation and is a result of a multicast event. The third type is (c) many-to-one, which occurs when many different components send an event type to a single component. The final type is (d) a combination of previous types (a)–(c), and is the most complex relation to resolve between multiple log formats.

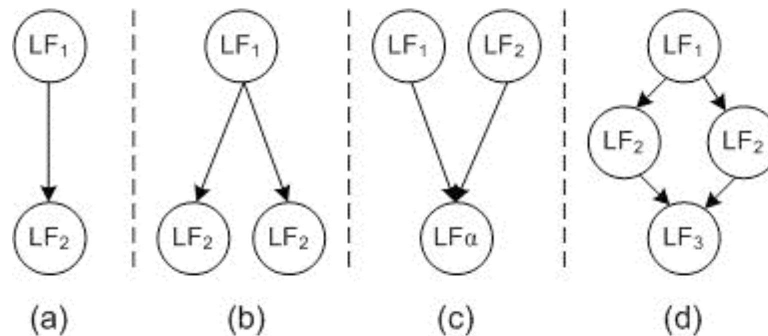

Figure 4. Four types of causal relations that can occur in a distributed system

```
1  procedure EVALUATE (dataflow, logmsgs) {
2   let dataflow' = topological_sort(dataflow);
3   let dataset = variables_table(dataflow);
4   let sorted = sort_host_then_time(logmsgs);
5
6   foreach (LF_i in dataflow') {
7     let K = C_i from CR_i,j
8
9     foreach (logmsg in sorted) {
10      if (logmsg matches LF_i) {
11        let V = values_of_variables(logmsg);
12
13        if (K is not empty set){
14          UPDATE dataset WITH V USING K;
15        }
16        else {
17          INSERT V INTO dataset;
18        }
19      }
20    }
```

```
21   }
22 }
```

Algorithm 1. General algorithm for evaluating dataflow models

If it is assumed that each entry in a message log contains its origin, e.g., hostname, then it is possible to use a dynamic programming algorithm and relational database theory (Atzeni & Antonellis, 1993) to reconstruct the data table that contains all the variables from a dataflow model. As shown in Algorithm 1, first a directed graph where log formats are nodes and the casual relations are edges is constructed. Next, the directed graph is topologically sorted so the evaluation knows the order to process each log format. This step is also necessary because when causal relation types (b) – (d) are present in the dataflow model specification, processing the log formats in reverse order of occurrence reduces algorithm complexity for constructing the dataset. Moreover, it ensures the algorithm has rows in the dataset to accommodate the data from log formats that occur prior to the current log format.

After topologically sorting the log formats, a dataset, which is a table that has a column for each log format variable in the dataflow model, is constructed. This dataset is constructed by first sorting the log messages by origin and time to ensure correct message sequence for each origin. More importantly, this enables presentation of the data trend over the lifetime of the system before aggregating the results.

Finally, each log format is matched against each log messages (or all the log messages for each log format is selected for processing). If there is a match, then the value of each variable in the log message is extracted, and the dataset is updated based on the following rules:

- **Rule for appending data.** If there is no cause variable set for the current log format, then the values from the log message are appended to the end of the data set.

- **Rule for inserting data.** If there is a variable set for the cause log format, then all the rows in the dataset where the cause's values equal the effect's values are updated (see Listing 3 for an example).

Finally, all incomplete rows are purged from the dataset and it is evaluated using the user-defined evaluation function (see Listing 4).

Managing 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, log formats are defined to identify variable portions of a message. The evaluation process is therefore expected to encounter similar log messages multiple times.

When constructing the data set in Algorithm 1, different variations of the same log format creates multiple rows in the final data set. QoS properties, however, are a single scalar value, and not multiple values. The following techniques are therefore used to address this concern:

- **Aggregation.** A function used to convert a dataset to a single value. Examples of an aggregation functions are, but not limited to: AVERAGE, MIN, MAX, and SUM.

- **Grouping.** Given an aggregation function, grouping is used to identify datasets that should be treated independent of each other. For example, in the case of causal relation (d) in Figure 4,

the values in the dataset for each sender (i.e., LF2) could be considered a group and analyzed independently.

Evaluating dataflow models in UNITE. UNITE implements Algorithm 1 using the SQLite relational database (www.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. Listing 3 shows an example SQL UPDATE query for inserting new data into the existing dataset based on causality relations between two log formats: LF1 and LF2.

```
UPDATE dataset SET LF1_recv = ?recv AND LF1_reqid = ?reqid
WHERE LF2_reqid = ?reqid;
```

Listing 3. SQL query for inserting new data into an existing dataset

Table 2 shows the variable table (or dataset) constructed by UNITE for the example dataflow model. 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 introduced at the latter part of the case study.

LF1.reqid	LF1.recv	LF2.reqid	LF2.reply
6	1234945638	6	1234945652
7	1234945690	7	1234945705
8	1234945730	8	1234945750

Table 2. A sample dataset for evaluating dataflow model in UNITE

Listing 4 shows the example evaluation function as an SQL query, which is used to evaluate the dataset in Table 2. The final result of this example would be 16.33 msec. Likewise, if the AVERAGE aggregation function is removed, then distributed system testers can view the data trend for average login time, which can help discover potential performance bottlenecks.

```
SELECT AVG(LF2.reply − LF1.recv) AS result FROM dataset;
```

Listing 4. SQL query for calculating average login time

Managing the Complexity of Dataflow Models

UNITE uses dataflow models to validate distributed system QoS properties. Although dataflow models enable UNITE to validate QoS properties independent of a SEM tool's capabilities, as dataflow models increase in size (i.e., number of log formats and relations between log formats) it becomes harder for distributed 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 (Harel, 1987).

To ensure efficient and effective application of dataflow models towards validating enterprise distributed system QoS attributes, UNITE leverages a model-driven engineering technique called domain-specific modeling languages (Sztipanovits & Karsai, 1997; Gray et al., 2007). Domain-specific modeling languages 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, domain-specific modeling languages 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. An invalid understanding of dataflow can result in an invalid specification of a dataflow model. By using domain-specific modeling languages, distributed system testers can construct dataflow models as graphs, which help visualize dataflow and ensure valid construction of such models, especially as they 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, distributed 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 domain-specific modeling languages, it is possible to enforce constraints that will ensure such relations are not possible in constructed models.

UNITE implements several Domain-specific modeling languages using an MDE tool called the Graphical Modeling Environment (GME) (Ledeczi, et al., 2001). GME allows system and software engineers, such as distributed system developers and testers, to author Domain-specific modeling languages for a target domain, such as dataflow modeling. End-users then construct models using the specified domain-specific modeling language and use model interpreters to generate concrete artifacts from constructed models, such as a configuration file that specifies how UNITE should evaluate a dataflow graph.

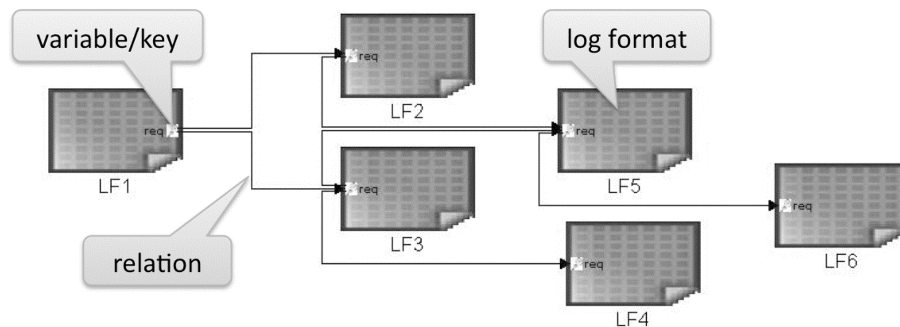

Figure 5. Example dataflow model in UNITE's domain-specific modeling language

Figure 5 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. The lines between two log formats represent a relation between variables in either log format. When distributed system testers create a relation between two different variables, the domain-specific modeling language validates the connection (i.e.*,* ensures the variable types are equal). Likewise, distributed system testers can execute the GME constraint checker to validate systemic constraints, such as validating that the

dataflow model is acyclic.

After constructing a dataflow model using UNITE's domain-specific modeling language, distributed system testers use model interpreters to auto-generate configuration files that dictate how to mine system execution traces. The configuration file is a dense XML-based file that would be tedious and error-prone to create manually. UNITE's domain-specific modeling language graphic representation and constraint checking therefore reduces complexity in managing dataflow models, thereby addressing Limitation 3 introduced at the latter part of the QED case study.

EXPERIMENTAL RESULTS

As mentioned in motivational case study, the QED project is in the early phases of it software lifecycle. 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 are using CUTS and UNITE to perform early integration testing. All tests were run in a representative testbed at ISISlab (www.isislab.vanderbilt.edu), which is powered by Emulab software (Ricci et al., 2003). 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 (Hill & Gokhale, 2007). Each scenario was designed so that all components communicate with each other using a single server in the GIG (similar to Figure 3). The first scenario was designed to test different thresholds of the underlying GIG middleware to discover 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.

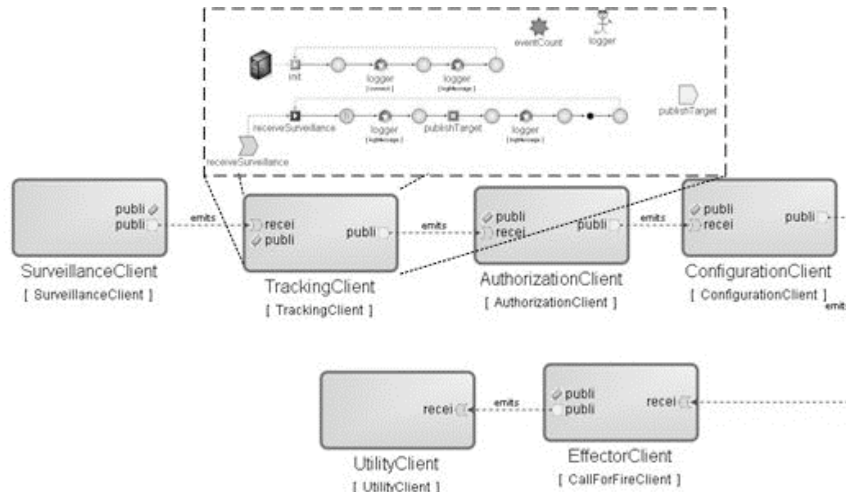

Figure 6. CUTS model of the multi-stage workflow test scenario

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. 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/workload (based on Timed I/O Automata (Kaynar, Lynch, Segala, & Vaandrager, 2006)) that dictates its actions during a test. Moreover, each behavior model contains actions for logging key data needed to validate QoS properties using dataflow models, similar to Listing 1 in the previous section. Listing 5 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 5. 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 preserve data integrity when data mining the system execution trace. QED developers next used UNITE to construct log formats (as discussed in the previous section) for identifying log messages during a test run that contain metrics of interest. These log formats were also used to define dataflow models that validate QED's QoS properties. In particular, QED developers were interested in validating and understanding the following QoS properties 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 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 a third-party vender provides the GIG middleware, 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 1 to calculate how much time the event spends in the server assuming event transmission is instantaneous, i.e., negligible.

$$(end_e - start_e) - \sum_c S_{c_e} \qquad (1)$$

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}.

Analyzing Multiple Publisher Results. Table 3 presents the results for tests that evaluate

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 this scenario was tested 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.

Publisher Name	Importance	Average E2E Response Time (msec)
Client A	30	103931.14
Client B	15	103885.47
Client C	10	103938.33

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

UNITE also allows QED developer and testers to view the data trend for the dataflow models QoS validation of this scenario to get a more detailed understanding of performance. Figure 7 shows how the response time of the event increases over the lifetime of the experiment. It is known beforehand that 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.

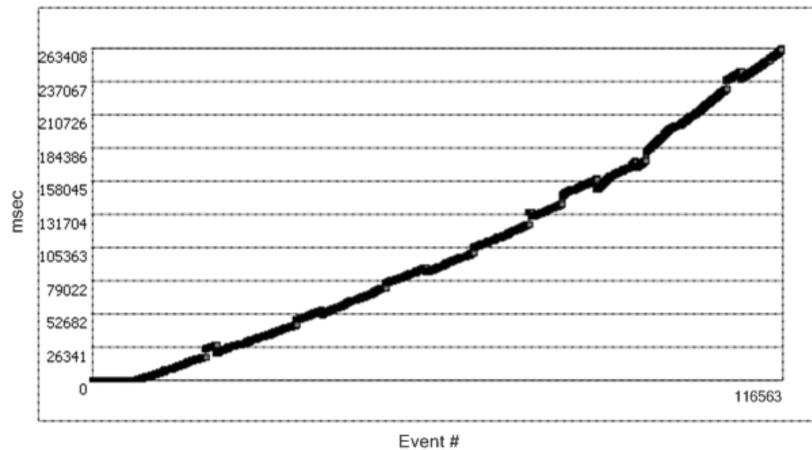

Figure 7. Data trend graph of average end-to-end response time for multiple publishers sending events at 75 Hz

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 queuing of events. Figure 8 graphs the data trend for the test, which is calculated by specifying Equation 1 as the evaluation for the dataflow model, and was produced by UNITE after analyzing (i.e., data mining metrics form) system execution traces. The test also consisted of several different log formats and causal relations, which were of types (a) and (b), as illustrated in Figure 4.

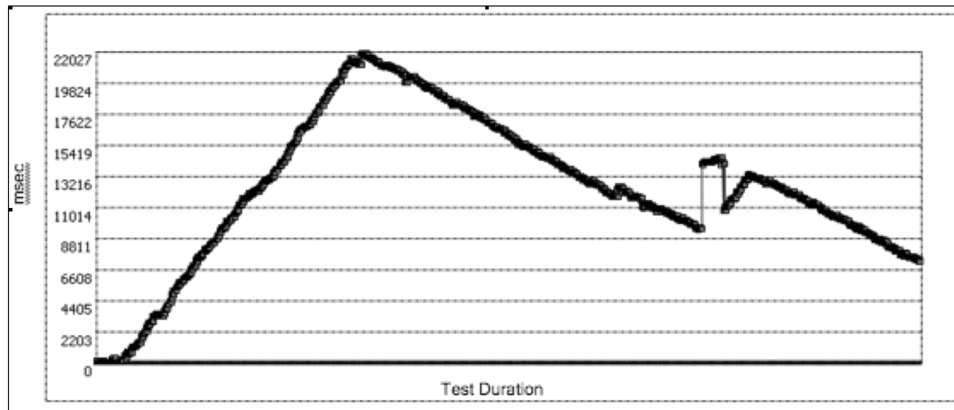

Figure 8. Data trend of the system placed in near optimal publish rate

Figure 8 shows the sustainable publish rate of the multi-stage workflow in ISISlab. This figure shows how the Java just-in-time 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 , 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 with comprehensive testing and analysis features using dataflow models to mine system execution traces via UNITE helped guide the development team to the next phase of testing and integration of feature sets.

FUTURE RESEARCH DIRECTIONS

Based on the results and experience developing and applying UNITE to a representative distributed system, the following is a list of future research directions:

- **Investigating techniques to optimize data mining and evaluation time.** As the system execution traces increase in size, the evaluation time of the dataflow model increases. Future work therefore should investigate techniques for optimizing evaluation of dataflow models so evaluation time is not dependent on the size of the system execution traces.

- **Investigating techniques for enabling multiple viewpoints (or aspects).** Creating system execution traces can be an expensive process because it requires executing the system in its target environment. Currently, a single dataflow graph is used to mine a system execution trace and used to validate a single QoS property. QoS, however, is a multi-dimensional property. Future research therefore should investigate techniques for enabling multiple viewpoints using a single dataflow model and system execution trace.

- **Investigating techniques to use dataflow models to mine validate the distributed system state using system execution traces.** System execution traces not only capture metrics, but it also captures the state of the system. Future research therefore should investigate techniques for validating system state while the system is both online (i.e., in real-time) and offline (i.e., after the system is shutdown) using system execution traces.

- **Investigating techniques to mine system execution traces and auto-construct dataflow models.** Although UNITE's domain-specific modeling language 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. This is because there is disconnect between the log messages used to generate execution traces and log formats that extract metrics from log messages in a system execution trace. Future research therefore should investigate techniques for auto-generating dataflow models from system execution traces to ease the specification process.

CONCLUSION

This chapter describes and evaluates a technique of using dataflow models to mine system execution traces and validate QoS properties. The chapter also describes how the dataflow and data mining techniques has been realized in a tool called UNITE. UNITE enables distributed system testers to validate QoS properties irrespective of SEM tool of choice. Moreover, UNITE can be used to validate QoS properties irrespective of the SEM tool's existing analytical capabilities.

Based on the results and experience developing and applying UNITE to a representative enterprise distributed system, the following lessons were learned:

- **Dataflow modeling increases the level of abstraction for validating QoS properties.** Instead of requiring knowledge of system composition and implementation, dataflow models provided a platform-, architecture-, and technology-independent technique for validating QoS properties.

- **Domain-specific modeling languages help manage the complexity of dataflow models.** This is because the domain-specific modeling languages provide QED testers with visual abstractions that were clear representations of the target domain. It therefore made it easier for them to compose such models, and ensure they were valid before using them to mine system execution traces.

REFERENCES

Ammons, G., Bodik, R., & Larus, J. R. (2002). Mining Specifications. *ACM SIGPLAN Notices, 37* (1), 4 – 16.

Atzeni, P., & Antonellis, V. D. (1993). *Relational Database Theory.* Redwood, CA, USA: Benjamin-Cummings Publishing Co.

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative Evaluation of Software Quality. *The 2nd International Conference on Software Engineering* (pp. 592-605). San Francisco, CA: IEEE Computer Society Press.

Chang, F., & Ren, J. (2007). Validating System Properties Exhibited in Execution Traces. *IEEE/ACM International Conference on Automated Software Engineering* (pp. 517-520). Atlanta, GA: ACM.

Denmat, T., Ducasse, M., & Ridoux, O. (2005). Data Mining and Cross-checking of Execution Traces: A Re-interpretation of Jones, Harrold and Stasko Test Information Visualization. *20th IEEE/ACM International Conference on Automated Software Engineering* (pp. 396 – 399). Long Beach, CA: ACM/IEEE.

Denton, T., Jones, E., Srinivasan, S., Owens, K. & Buskens, R.W. (2008). NAOMI – An Experimental Platform for Multi-modeling. *ACM/IEEE 11th International Conference on Model Driven Engineering Languages & Systems*. Toulouse, France.

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

El-Gendy, M. A., Bose, A., & Shin, K. (2003). Evolution of the Internet QoS and Support for Soft Real-time Applications. *Proceedings of the IEEE , 91* (7), 1086-1104.

Gray, J., Tolvanen, J.-P., Kelly, S., Gokhale, A., Neema, S., & Sprinkle, J. (2007). Domain-Specific Modeling. In P. Fishwick, *CRC Handbook on Dynamic System Modeling* (pp. 7.1-7.20). CRC Press.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming , 8* (3), 231-274.

Heineman, G. T., & Councill, W. T. (2001). *Component-based Software Engineering: Putting the Pieces Together*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

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

Hill, J. H., Schmidt, D. C., Slaby, J., & Porter, A. (2008). CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments. *15th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems* (pp. 66-75). Belfast, Northern Ireland: IEEE Computer Society.

Hill, J. H., Slaby, J. M., Baker, S., & Schmidt, D. (2006). Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. *12th International Conference on Embedded and Real-Time Computing Systems and Applications* (pp. 350-362). Sydney, Australia: IEEE Computer Society.

Jilani, A. A., Nadeem, A., Kim, T.-h., & Cho, E.-s. (2008). Formal Representations of the Data Flow Diagram: A Survey. *Advanced Software Engineering and Its Applications*, 153-158.

Jones, J. A., Harrold, M. J., & Stasko, J. (2002). Visualization of Test Information to Assist Fault Localization. *24th International Conference on Software Engineering* (pp. 467 – 477). Orlando, FL: ACM.

Joukov, N., Wong, T., & Zadok, E. (2005). Accurate and Efficient Replaying of File System Traces. *4th Conference on USENIX Conference on File and Storage Technologies* (p. 25). San Francisco, CA: USENIX Association.

Kaynar, D. K., Lynch, N., Segala, R., & Vaandrager, F. (2006). *The Theory of Timed I/O Automata.* San Rafael, CA, USA: Morgan and Claypool Publishers.

Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., & Sprinkle, J. (2001). Composing Domain-Specific Design Environments. *IEEE Computer , 34* (11), 44-51.

Ledeczi, A., Maroti, M., Karsai, G., & Nordstrom, G. (1999). Metaprogrammable Toolkit for Model-Integrated Computing. *the IEEE International Conference on the Engineering of Computer-Based Systems Conference* (pp. 311-). Nashville, TN: IEEE Computer Society.

Lee, E. A., & Parks, T. M. (2002). *Dataflow Process Networks.* Norwell, MA, USA: Kluwer Academic Publishers.

Lo, D., & Khoo, S. (2006). SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner. *14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 265 – 275). Portland, OR: ACM.

Lo, D., Maoz, S., & Khoo, S. (2007). Mining Modal Scenario-based Specifications from Execution Traces of Reactive Systems. *22nd IEEE/ACM International Conference on Automated Software Engineering* (pp. 465 – 468). Atlanta, GA: IEEE/ACM.

Lo, D., Khoo, S., & Liu, C. (2008). Mining Past-time Temporal Rules from Execution Traces. *International Workshop on Dynamic Analysis* (pp. 50 – 56). Seattle, WA: ACM.

Loyall, J., Carvalho, M., Schmidt, D., Gillen, M., Martignoni III, A., & Bunch, L. (2009). QoS Enabled Dissemination of Managed Information Objects in a Publish-Subscribe-Query Information Broker. *Defense Transformation and Net-Centric Systems.* Orlando, FL.

Mania, D., Murphy, J., & McManis, J. (2002). Developing Performance Models from Nonintrusive Monitoring Traces. *IT & T* .

Mann, J. (1996). *The Role of Project Escalation in Explaining Runaway Information Systems Development Projects: A Field Study.* Georgia State University, Atlanta, GA.

Moe, J., & Carr, D. A. (2001). Understanding Distributed Systems via Execution Trace Data. *9th International Workshop on Program Comprehension* (pp. 60). Toronto, Canada: IEEE Computer Society.

Mos, A. M., & Murphy, J. (2001). Performance Monitoring of Java Component-Oriented Distributed Applications. *9th International Conference on Software, Telecommunications and Computer Networks*, (pp. 9-12). Dubrovnik, Croatia.

National Security Agency. (2009, June 28). *Global Information Grid*. Retrieved August 5, 2009 from National Security Agency: http://www.nsa.gov/ia/programs/ global_industry_grid/ index. shtml

Parsons, T., Mos, A., & Murphy, J. (2006). Non-Intrusive End-to-End Runtime Path Tracing for J2EE Systems. *IEE Proceedings-Software* .

Pezzini, M., & Natis, Y. V. (2007). *Trends in Platform Middleware: Disruption Is in Sight* . Retrieved June 2008, from Gartner: www.gartner.com/DisplayDocument?doc_cd=152076

Ricci, R., Alfred, C., & Lepreau, J. (2003). A Solver for the Network Testbed Mapping Problem. *SIGCOMM Computer Communications Review , 33* (2), 30-44.

Rittel, H., & Webber, M. (1973). Dilemmas in a General Theory of Planning. *Policy Sciences , 4* (2), 155-169.

Russell, N., van der Aalst, W. M., ter Hofstede, A. H., & Wohed, P. (2006). On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. *3rd Asia-Pacific Conference on Conceptual Modelling* (pp. 95 – 104). Hobart, Australia: Australian Computer Society, Inc.

Schmidt, D. C. (2006). Model-Driven Engineering. *IEEE Computer , 39* (2).

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

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

Snow, A., & Keil, M. (2001). The Challenges of Accurate Project Status Reporting. *34th Annual Hawaii International Conference on System Sciences.* Maui, Hawaii: ACM.

Software Engineering Institute. (2006). *Ultra-Large-Scale Systems: Software Challenge of the Future.* Carnegie Mellon University. Pittsburgh, PA: Carnegie Mellon.

Sztipanovits, J., & Karsai, G. (1997). Model-Integrated Computing. *IEEE Computer , 30* (4), 110-112.

Tortonesi, M., Stefanelli, C., Suri, N., Arguedas, M., & Breedy, M. (2006). Mockets: A Novel Message-Oriented Communications Middleware for the Wireless Internet. *International Conference on Wireless Information Networks and Systems.* Setubal, Portugal.

Vazquez, F. (1994). Identification of Complete Dataflow Diagrams. *SIGSOFT Software Engineering Notes, 19* (3), pp. 36 – 40.

Woodside, M., Franks, G., & Petriu, D. C. (2007). The Future of Software Performance Engineering. *The Future of Software Engineering* (pp. 171 – 187). Minneapolis, MN.

ADDITIONAL READING

Chatterjee, A. (2007). Service-component architectures: A programming model for SOA. *Dr. Dobb's Journal, 400*, 40 – 45.

Chilimbi, T. M., & Hauswirth, M. (2004). Low-overhead memory leak detection using adaptive statistical profiling. *Proceedings of the 11th international Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA.

Cascaval, C., Duesterwald, E., Sweeney, P. F., & Wisniewski, R. W. (2006). Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development, 50* (2/3), 239 – 248.

Haran, M., Karr, A., Orso, A., Portor, A., & Sanil, A. (2005). Applying classification techniques to remotely-collected program execution data. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal.

Hauswirth, M., Sweeney, P., Diwan, A., & Hind, M. (2004). Vertical profiling: Understanding the behavior of object-oriented applications. *ACM SIGPLAN Notices, 39* (10), 251 – 269.

Huselius, J., & Andersson, J. (2005). Model synthesis for real-time systems. *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, Manchester, UK.

Laugelier, G., Sahraoui, H., & Poulin, P. (2005). Visualization-based analysis of quality for large-scale software systems. *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering,* Long Beach, CA.

Li, Z., Sun, W., Jiang, Z. B., & Zhang, X. (2005). BPEL4WS unit testing: Framework and implementation. *Proceedings of the IEEE International Conference on Web Services*, Orlando, FL.

Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P., & Maroti, M. (2001). On metamodel composition. *Proceedings of the 2001 IEEE International Conference on Control Applications*, Mexico City, Mexico.

Kounev, S., & Buchmann, A. (2003). Performance modeling and evulation of large-scale J2EE applications. *Proceedings of the 29th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems*, Dallas, TX.

Kounev, S. (2006). Performance modeling and evaluation of distributed component-based systems using queuing Petri nets. *IEEE Transactions of Software Engineering, 32* (7), 486 – 502.

Memon, A., Porter, A., Nagarajan, A., Schmidt, D., & Natarajan, B. (2004). Skoll: Distributed quality assurance. *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland.

Metz, E., Lencevicius, R., & Gonzalez, T. (2005). Performance data collection using a hybrid approach. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal.

Mos A., & Murphy, J. (2004). COMPAS: Adaptive Performance Monitoring of Component-Based Systems. *Proceedings of 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems*, Beijing, China.

Odom, J., Hollingsworth, J. K., DeRose, L., Ekanadham, K., & Sbaraglia, S. (2005). Using dynamic tracing sampling to measure long running programs. *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Seattle, WA.

Parsons, T. & Murphy, J. (in press). Detecting performance antipatterns in component-based enterprise systems. *Journal of Object Technology*.

Saff, D., & Ernst, M. D. (2004). An experimental evaluation of continuous testing during development. *Proceedings of the 2004 ACM SIGSOFT international Symposium on Software Testing and Analysis*, Boston, MA.

Schroeder, P. J., Kim, E., Arshem, J., & Bolaki, P. (2003). Combining behavior and data modeling in automated test case generation. *Proceedings of the 3rd International Conference on Quality Software*, Dallas, TX.

Srinivas, K., & Srinivasan, H. (2005). Summarizing application performance from a components perspective. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal.

Stewart, C., & Shen, K. (2005). Performance modeling and system management for multi-component online services. *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA.

Wu, W., Spezialetti, M., & Gupta, R. (1996). Designing a non-intrusive monitoring tool for developing complex distributed applications. *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems*, Washington, D.C.

KEY TERMS AND DEFINITIONS

Key Terms

enterprise distributed systems, dataflow modeling, system execution traces, quality-of-service validation, system execution modeling, log formats, casual relations, domain-specific modeling language

Definitions

Enterprise distributed systems are systems characterized to be large at-scale, consists of many software components deployed on many hardware resources, and communicate via network to accomplish different operational scenarios.

Dataflow modeling is the process of identifying and modeling how data moves around an information system, such as an enterprise distributed system.

System execution trace is a collection (or sequence) of text-based messages that capture the behavior and state of an application, such as an enterprise distributed system, throughout its execution lifetime.

Quality-of-service (QoS) validation is the process of evaluating quality-of-service (QoS) properties, such as end-to-end response time, throughput, and scalability, of a system on the target architecture.

Domain-specific modeling language is a modeling language that captures the semantics and constraints of a given domain and provides intuitive visual notations that enable end-users to easily construct valid models that realize concepts for the target domain.

System execution modeling is the process of using domain-specific modeling languages to model the behavior and workload of a system and use constructed models to validate different system properties, such as QoS properties.

Log formats are high-level constructs that capture both constant and variable portions of individual, yet similar, log messages in a system execution trace.

Causal relations are relations that define how data in a dataflow model relates across different (application) contexts.