Adapting System Execution Traces for Validation of Distributed System QoS Properties

T. Manjula Peiris and James H. Hill
Dept. of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
Email: {tmpeiris, hillj}@cs.iupui.edu

***Abstract*—System execution traces are useful artifacts for validating distributed system quality-of-service (QoS) properties, such as end-to-end response time, throughput, and service time. With proper planning during development phase of the software lifecycle, it is possible to ensure such traces contain required properties to facilitate analysis for QoS validation. In some case, however, it is not possible to ensure system execution traces contain the necessary properties for QoS analysis.**

This paper presents the *System Execution Trace Adaptation Framework (SETAF)* for adapting system execution traces to support analysis of QoS properties. It also presents results from applying SETAF to externally developed applications. The results show that it is possible to validate QoS properties by automatically adapting system execution traces at analysis time instead of modifying the application's existing source code.

***Keywords*-SETAF, QoS validation, system execution traces, adaptation**

I. Introduction

System execution traces [1], which are a collection of log messages, are useful artifacts for analyzing distributed system quality-of-service (QoS) properties, such as end-to-end response time, service time and throughput. One benefit of using system execution traces for QoS validation is that they provide a comprehensive view of the system's behavior and state throughout its execution lifetime. This is opposed to a single snapshot of the system at a given point in time, such as a global snapshot [2]. Likewise, they provide distributed system testers with a rich set of data for analyzing data trends associated with a given QoS property, *i.e.*, how a given QoS property changes with respect to time.

The UNITE [3] tool describes a method for using system execution traces to validate distributed system QoS properties. UNITE accomplishes this feat by applying relational database theory [4] techniques and dataflow models to analyze different QoS properties. For example, a system execution trace may contain messages for the sending/receiving of an event along with a timestamp for each occurrence of the message. By searching for the send/receive message keywords and using a dataflow model to show their relation, it is possible to mine the system execution trace for such messages and analyze event latency between the sender and receiver using a general-purpose tool. More importantly, such analysis is possible irrespective of system complexity, composition, and implementation because the dataflow model is at a higher level-of-abstraction than the concrete system and system execution traces are platform-, technology-, and language-independent artifacts.

Although it is possible to validate QoS properties via system execution traces, system execution traces must contain several properties, *e.g.*, identifiable keywords, to support QoS analysis. Moreover, the dataflow model used to analyze the system execution trace must contain several properties, *e.g.*, identifiable log formats and unique relations between different log formats. If planned early enough in the software lifecycle, it is possible to ensure these properties exist in both the dataflow model and generated system execution trace. Unfortunately, it is not possible to *always* ensure that system execution traces contain properties required for analysis and QoS validation. This is because system execution traces are primarily used for debugging functional software problems [5].

In software design, the adapter [6] software design pattern presents the concept of adapting one object (or abstraction) so it has the expected interface another object expects. In the context of analyzing system execution traces, the system execution traces and dataflow model can be classified as the *adaptee* and UNITE can be classified as the *client*. The main challenge, however, is determining how either the system execution trace and dataflow model must be adapted so that either contains the necessary properties to support analysis and QoS validation.

This paper therefore presents a method for adapting system execution traces and their dataflow models to contain properties required for QoS validation. The main contributions of this paper are as follows:

- It presents *System Execution Trace Adaptation Framework (SETAF)*, which is a framework for adapting system execution traces and dataflow models so they contain the required properties for analysis and QoS validation;
- It is the first attempt, to the best of the authors knowledge, as a method and a tool for automatically adapting system execution traces to support QoS analysis if they do not contain the necessary properties to undergo such analysis;
- It discusses different design alternatives—including their advantages and disadvantages—for adapting system execution traces for QoS analysis; and
- It showcases how SETAF can be applied to system

This works was sponsored in part by the Australian Defense Science and Technology Organization (DSTO).

TABLE I
AN EXAMPLE SYSTEM EXECUTION TRACE DISPLAYED IN TABLE FORMAT AS IF BEING STORED FOR OFFLINE ANALYSIS IN A DATABASE.

Time of Day	Hostname	Severity	Message
2011-02-25 05:15:55	sender.csilab.vanderbilt.edu	INFO	Config: sent event 5 at 120394455
2011-02-25 05:15:55	sender.csilab.vanderbilt.edu	INFO	Planner: sent event 6 at 120394465
2011-02-25 05:15:55	receiver.csilab.vanderbilt.edu	INFO	Planner: received event 5 at 120394476
2011-02-25 05:15:55	sender.csilab.vanderbilt.edu	INFO	Config: sent event 7 at 120394480
2011-02-25 05:15:55	receiver.csilab.vanderbilt.edu	INFO	Effector: received event 6 at 120394488
2011-02-25 05:15:55	receiver.csilab.vanderbilt.edu	INFO	Planner: received event 7 at 120394502

execution traces generated by different systems.

Experimental results from applying SETAF to several open-source software projects show that it is able to adapt system execution traces without requiring *any* modification to the original source code that generates the system execution traces.

Paper organization. The remainder of this paper is organized as follows: Section II provides a detailed overview of UNITE and its current limitations; Section III presents the design and methodology of SETAF; Section IV presents the results of applying SETAF to several open source projects; Section V discusses work related to SETAF; and Section VI presents concluding remarks and future research directions.

II. OVERVIEW OF UNITE

UNITE is a method and tool for analyzing system execution traces and validating QoS properties of distributed systems. This is because the execution traces from such systems are *hard* to analyze due to their size and complexity. UNITE, however, can perform such analysis in $O(n)$ runtime complexity irrespective of the target system execution trace, which is *hard* to do using simple scripts that are typically hard-coded for a specific use case (or system execution trace). Because UNITE is not a well-known tool, the remainder of this section provides a detailed overview of UNITE's capabilities and its current limitations.

Table I presents an example system execution trace from a distributed system. As shown in this table, the system execution trace has log messages corresponding to send and receive events that occur between components in a distributed system. The log messages in this example contain time of the event, event id, and the component name where the event occurred.

Distributed system testers use UNITE to analyze QoS properties, such as latency, via the system execution trace in Table I by first identifying what log messages need to be extracted from the execution trace. These log messages should contain metrics of interest that will be used to perform the QoS analysis.

Once the log messages have been identified, distributed system testers convert the common log messages to *log formats*. A log format is a high-level representation of a log message that captures both the static and variable portions of its corresponding log message. The static portions are those that do not change between different log messages, which is an instance of a log format. The variable portions are those that change between different instances of a log format.

Given the system execution trace in Table I, Listing 1 shows the log formats for extracting its log messages. As shown in this listing, each log format (*e.g.*, `LF1` and `LF2`) contains static and variable portions for extracting metrics from its corresponding log message in the system execution trace. For example, `LF1` contains the variables: `cmpid`, `eventid`, and `sent`. The `sent` variable is used to extract the sending time. The remaining variables in the log format are used for correlating messages, which is explained next.

```
LF1: {STRING cmpid} sent event {INT eventid}
 at {INT sent}

LF2: {STRING cmpid} received event {INT eventid}
 at {INT recv}

 Relation:
  LF1.cmpid = LF2.cmpid
  LF1.eventid = LF2.eventid
```

Listing 1. Dataflow model for analyzing system execution trace.

After defining the log formats for extracting metrics of interest from a system execution trace, distributed system testers define a dataflow model that captures the relationship between the different log formats. Like dataflow models in program analysis where dataflow models relate variables across different source lines, dataflow models in UNITE relate log format variables across different log messages (or application contexts). The dataflow model enables reconstruction of execution flows in the system (1) irrespective of system complexity and composition and (2) without a need for a global clock to ensure causality [2]. This is because the relations between the log formats preserve causality.

```
AVG(LF2.recv - LF1.sent)
```

Listing 2. Expression for analyzing event latency using UNITE.

Finally, distributed system testers define an expression that validates a given QoS property based on the variables in the log format. For example, Listing 2 highlights the expression for evaluating average event latency. UNITE then uses this dataflow model and expression to mine the system execution trace and evaluate the provided expression. Likewise, if the aggregation function (*i.e.*, `AVG`) is removed from the expression, then UNITE will present the data trend for the QoS property that is targeted in the analysis.

A. Current Limitations of UNITE

Although distributed system testers can use UNITE to analyze distributed system QoS properties via system execution traces, UNITE's method has several limitations, which we highlight as the following challenges:

TABLE II
EXAMPLE SYSTEM EXECUTION TRACE THAT DOES NOT CONTAIN UNIQUE IDS.

Time of Day	Hostname	Severity	Message
2011-02-25 12:00:55	sender.csilab.vanderbilt.edu	INFO	Started doing task A at 12.00
2011-02-25 12:01:55	sender.csilab.vanderbilt.edu	INFO	Finished doing task A at 12.01
2011-02-25 12:02:55	receiver.csilab.vanderbilt.edu	INFO	Started doing task A at 12.02
2011-02-25 12:03:55	sender.csilab.vanderbilt.edu	INFO	Finished doing task A at 12.03

- **Challenge 1: Correlating log formats that have non-unique instances.** UNITE assumes that any instance of a particular log format is different from any other instance of the same log format. As shown in Table I and Listing 1, the event ids are different in different log formats. It, however, is possible for the same log message to reoccur without a unique id. When this happens, the relation between the two log messages is considered *non-unique*. Consequently, analysis of a system execution trace with non-unique relations typically yields incorrect results.
  For example, Table II illustrates an example system execution trace where the different instances of the same log format are similar. Only the variable parts related to time changes in different instances. It is therefore *hard* to know what two start/finish messages are associated with each other without human intervention. Moreover, when an example similar to the one present in Table II is analyzed by UNITE, it will yield incorrect results. It is therefore critical that UNITE be able to handle such situations in generated system execution traces.
- **Challenge 2: Correlating log formats with hidden relations.** System execution traces typically capture a variety of events that occur in different software components. When there are repetitive events as shown in Table I, it is easy to identify the relations between log formats. In other cases, there may be no repetitive events in the system. When this occurs, there are no *true* variable parts (other than the log message time) for defining causality between log formats. When this occurs, we say the dataflow model and system execution trace contains *hidden relations*.
  For example, consider the system execution trace in Table III. Time is a variable in each log format and each log message is unique, however, there is no explicit variable for determining causality between the log messages. The system execution trace in Table III therefore cannot be analyzed using UNITE, but it is critical that UNITE be able to handle such situations.
- **Challenge 3: Associating values of newly added log format variables.** One of UNITE's main assumptions is that values for a given log format variable are populated using data from its corresponding log messages. Correlating log formats in Table II and Table III, however, requires adding new log format variables in the dataflow model while preserving the relationship between different log formats. This process is sometimes as simple as adding a monotonically increasing id. Other times it requires coordinating values from other log messages. There is no uniform way to associate data for the newly added log format variables, but UNITE must be able to handle such situations.

The challenges listed above illustrate the heterogeneity among different system execution traces in distributed systems. Although system execution traces vary from system to system, it is possible to use a general-purpose approach for adapting them to support QoS analysis. The next section therefore explains how SETAF addresses the challenges outlined above to enable QoS analysis of system execution traces.

III. The Design and Functionality of SETAF

This section describes the design and functionality of SETAF. This section also uses concrete examples to illustrate concepts realized in SETAF.

A. Design Approaches for Adapting System Execution Traces

Before discussing the details of SETAF, it is necessary to understand and discuss different approaches that can be used to adapt system execution traces and dataflow models for QoS analysis, in particular with UNITE. The following therefore is a list of approaches for realizing the adaptation:

- **Approach 1: Change system execution traces directly.** Because lack of certain properties in system execution traces is preventing valid QoS analysis, the system execution traces can be changed according to the dataflow model defined in UNITE. For example, the underlying source code that generates the trace can be changed such that each log format has variables for capturing unique relations. The advantage of this approach is that UNITE can directly analyze the generated system execution traces.
  This approach, however, has several disadvantages. First this approach requires distributed system testers to have access to the source code so they can make the necessary updates. Moreover, it also requires distributed system testers to be familiar with the source code—and its implementation—to make the necessary updates. Secondly, updating the source code accordingly can be a costly, error prone, and time consuming task—especially when dealing with a very large code base. Finally, the actual source code should not be changed just to validate its QoS properties because such changes may affect the functional and non-functional requirements of the system.
- **Approach 2: Adapting the dataflow model inside UNITE.** It is possible to adapt the dataflow model directly by modifying UNITE's source code. For example, if the adaptation requires adding a new log format variable and a relation, then UNITE could be updated to add new variables to an existing dataflow model.

TABLE III
EXAMPLE SYSTEM EXECUTION TRACE THAT CONTAIN HIDDEN RELATIONS.

Time of Day	Hostname	Severity	Message
2011-02-25 10:00:55	sender.csilab.vanderbilt.edu	INFO	Initializing the system at 10.00
2011-02-25 10:10:55	sender.csilab.vanderbilt.edu	INFO	Start Monitoring components at 10.10
2011-02-25 10:11:55	receiver.csilab.vanderbilt.edu	INFO	Finish Monitoring components at 10.11
2011-02-25 10:40:55	sender.csilab.vanderbilt.edu	INFO	Shutting down the system at 10.40

The advantage in this approach is that the source code of the actual distributed system doesn't need to be changed. Unfortunately, it is not possible to adapt each dataflow model in the same manner. This is because the dataflow model is associated with the given system that generates the system execution trace under QoS analysis. A dataflow model therefore can only be reused for different executions of the same system. Moreover, this implies that UNITE must be updated to accommodate new dataflow models that need adaptation.

- **Approach 3: Adapting the dataflow model using user-defined external adapters.** This is similar to Approach 2, *i.e.*, adapting the dataflow model inside UNITE, but now the mechanisms for adapting the dataflow model resides in an external specification. This external specification is loaded by UNITE when analyzing the corresponding system execution trace. This approach therefore allows distributed system testers to write their own adaptation specification according to the system domain without modifying the distributed system's existing source code. The disadvantage of this approach is that to create their own adaptation specification, distributed system testers should be aware of the limitations of the existing dataflow model. They also need to identify what are the new log format variables and the relations need to be added to the existing dataflow model so that it is to be adapted for the correct QoS analysis.

Based on the advantages and disadvantages of each approach discussed above, Approach 3 was selected as the approach for adapting system execution traces for analysis via UNITE. This approach was selected because it addresses the heterogeneity among different systems in different domains. Moreover, it provides greater flexibility and configurability when analyzing system execution traces because UNITE's underlying theory and algorithms can remain constant while allowing the adapter(s) to provide more domain-specific details.

B. Defining the Adaptation Specification

As discussed in the previous section, the approach of using external adapters was selected for adapting system execution traces for QoS analysis using UNITE. Distributed system testers use SETAF by first *manually* analyzing the generated system execution trace. Through this analysis, the tester identifies an adaptation pattern. The *adaptation pattern* captures what properties must be added to the dataflow model in order for UNITE to correctly analyze the system execution trace. Each adaptation pattern contains the following details:

- **Variables.** The variables are private data points that assist with adapting its corresponding system execution trace. The variables are visible only to the adapter pattern, and not visible to UNITE—thereby helping to address Challenge 3 introduced in Section II-A.
- **Data points.** The data points are new columns added to UNITE's data table for reconstructing valid system execution flows from the generated system execution trace. For example, a data point named `LF1.uid` will become a column name in UNITE's data table. Finally, the data points are used to create new relations in UNITE's data table—thereby addressing Challenge 1 introduced in Section II-A.
- **Relations.** The relations section of the adaptation pattern inserts new causality relations among log formats into the dataflow model. For example, assume the following two data points named `LF1.uid` and `LF2.uid` are added to the dataflow model. This section is used to define that `LF1.uid` causes `LF2.uid`—thereby addressing Challenge 2 introduced in Section II-A.
- **Adaptation code.** The adaptation code is where the domain-specific logic resides for the adaptation pattern. The adaptation code is segmented based on the log formats that must undergo adaptation. Each segment dictates how to update variables in the dataflow model, as well as its own private variables—thereby helping address Challenge 3 introduced in Section II-A.

Realization in SETAF. To show the adaptation specification (capturing an adaptation pattern) defined in SETAF, we are going to use a portion of an example system execution trace of Apache ANT (ant.apache.org), which is presented in Table II. We select Apache ANT because its adaptation specification is a simple example for illustrating the concepts previously discussed. We, however, have applied SETAF to more complex examples as explained in Section IV.

Apache ANT is a widely used build tool primarily for Java projects, but can be used for other purposes (*e.g.*, build automation, documentation generation, and traditional execution shell). Apache ANT completes different tasks during the build process. A task finish event is the effect of a task start event. Using this domain knowledge of the system execution trace, Listing 3 illustrates the dataflow model for analyzing the execution time of each task in Apache ANT.

```
LF1: Started doing task {STRING taskname}
 at {INT startTime}

LF2: Finished doing task {STRING taskname}
 at {INT finishTime}

Relation:
```

```
LF1.taskname = LF2.taskname
```

Listing 3. Dataflow model for Apache ANT system execution trace.

When repeating the same task, Apache ANT uses the same task name in different log messages, which will result in identical instances of `LF1` and `LF2` (*i.e.* differs only from time stamp) in the system execution trace. This is similar to the Challenge 1 describes in Section II. Although ANT system execution trace has this problem, log messages representing the start of a task are *always* preceded by a log message representing the completion of a task. This observation can be used to write a SETAF specification that adapts the dataflow model of Apache ANT system execution trace. Listing 4 highlights the adaptation pattern written as a SETAF specification to ensure correct analysis of Apache ANT's system execution trace.

```
Variables:
  int id_;

Init:
  id_ = 0;

DataPoints:
  int LF1.uid;
  int LF2.uid;

Relations:
  LF1.uid -> LF2.uid;

// Begin adaptation code section
On LF1:
  SETAF::int_vp (vars["uid"])->value (this->id_);
  this->id_++;

On LF2:
  SETAF::int_vp (vars["uid"])->value (this->id_);
  this->id_++;
```

Listing 4. Adaptation pattern specification for Apache ANT in SETAF.

As illustrated in this listing, first distributed system testers define the variables needed to assist in adapting the system execution trace. This information is captured in the section labeled *Variables* of the SETAF specification. Distributed system testers then use the *DataPoints* section to specify what data points need to be added to each log format. For example, two data points named `LF1.uid` and `LF2.uid`, which are of integer type, are injected into the dataflow model. These two variables are needed to ensure that the relations are unique between the two log formats named `LF1` and `LF2`.

After defining what data points need to be injected into the dataflow model, distributed system testers define new relations that should be added to the dataflow model. As illustrated in Listing 4, the left side of the arrow represents the cause variable; whereas, right side of the arrow represents effect variable. This specification of the relations is similar to how existing relations are defined in UNITE.

The final part of the SETAF specification is defining how to adapt the actual system execution trace. Distributed system testers accomplish this task by stating how the adapter transforms the system execution trace for each log format that needs adaptation. As shown in Listing 4, the `uid` variable is assigned the current value of `id_` in both `LF1` and `LF2`. In both `LF1` and `LF2` the state variable `id_` is incremented. This ensures the next occurrence of `LF1` is differentiated from the previous occurrence of `LF1`, as well as `LF2`.

Finally, the identifier `SETAF::int_vp ()` represents a log format variable casting operator. It is needed because all the variable types in UNITE are derived from a common variable type. This casting operator allows the distributed software tester to narrow the generic variable type to its concrete variable type, such as an integer, to set its value accordingly. SETAF has log format variable casting operators for each variable type supported in UNITE.

C. *Adaptation of the Dataflow Model at Analysis Time*

The previous section discussed how distributed system testers can define an adaptation pattern (*i.e.*, SETAF specification) to adapt dataflow models and system execution traces for QoS analysis. Given an adaptation pattern, the overall approach for adapting the dataflow model and system execution trace is described in the following steps:

1) Use the *data points* section of the adaptation pattern to update the dataflow model. Each data point becomes a new column in the underlying data table that represents the dataflow model in UNITE.
2) Use the *relations* section of the adaptation pattern to update the existing relations between existing log formats in the dataflow model.
3) For each log format encountered while processing the system execution trace, use the adaptation code to update variables in the reconstructed data table. The variables being updated by the adapter pattern are those that appear in the adapter pattern.

Realization in SETAF. Given a SETAF specification, SETAF transforms the specification into C++ source code, which is compiled into an external module. The external module is then loaded by UNITE to adapt its corresponding system execution trace for QoS analysis based on the algorithm discussed above. Figure 1 shows a high-level overview of UNITE and SETAF using the compiled adapter external module.

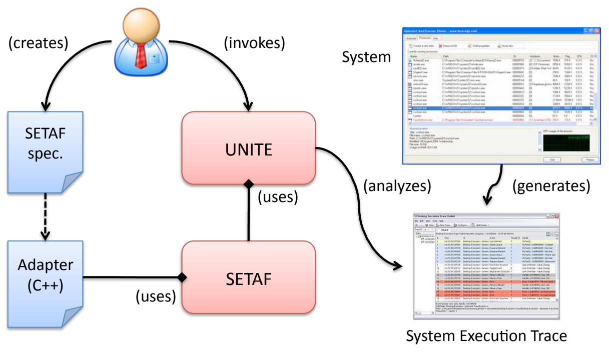

Fig. 1. Conceptual overview of SETAF's workflow.

```
void Ant_Adapter::init (void) {
  this->id_ = 0;
}
```

```
void Ant_Adapter::reset (void) {
  this->id_ = 0;
}

void Ant_Adapter::
update_log_format (Log_Format * lfmt) {
  const string & name = lfmt->name ();

  if (name == "LF1")
    lfmt->add_variable ("uid", "int");
  else if (name == "LF2")
    lfmt->add_variable ("uid", "int");
}

void Ant_Adapter::
update_relation (Log_Format * lfmt) {
  const string & name = lfmt->name ();

  if (name == "LF1")
    lfmt->add_relation ("LF2", "uid", "uid");
}

bool Ant_Adapter::
update_variables (Variable_Table & vars,
                  Log_Format * lfmt) {
  const ACE_CString & name = lfmt->name ();

  if (name == "LF1") {
    SETAF::int32_vp (vars["uid"])->value (this->id_);
        ++this->id_;
  }
  else if (name == "LF2") {
    SETAF::int32_vp (vars["uid"])->value (this->id_);
    ++this->id_;
  }

  return true;
}
```

Listing 5. Auto-generated source code for Apache ANT adapter.

Listing 5 showcases the source code auto-generated for the Apache ANT adapter used by SETAF that appears in Listing 4. As shown in this listing, the variables in the *Variables* section of the SETAF specification are mapped into private variables in the adapter. Likewise, the *DataPoints* in the specification are used inside the `update_log_format()` function. More specifically, these data points are used to create new log format variables.

Similarly, the `update_relation()` function uses the relations specified in the *Relations* section of the specification. This function is therefore responsible for creating new relations among log formats with respect to the new log format variables. The `update_variables()` function does the actual adaptation. Each adaptation section in the SETAF specification (*i.e.*, `On [name]`) is given its own if statement based on the log format's unique name provided as defined in the dataflow model. Lastly, UNITE uses SETAF to adapt its analysis of the system execution trace using the specified adapter.

IV. Results for Applying SETAF to open source projects

This section presents results from applying SETAF to several open-source applications and systems that generate system execution traces without the properties required for QoS analysis using UNITE.

A. Experimental Setup

To determine applicability and validity of SETAF's technique, we applied SETAF to the following open-source applications and systems:

- **Apache ANT.** Apache ANT, which was previously introduced in Section III-B, is a widely used java library and a command line tool mainly used to build Java-based software systems.
- **Apache Tomcat Web Server.** Apache Tomcat (tomcat.apache.org) is an implementation of the Java Servlet [7] and JavaServer Pages [8] technology. It is also one of the most widely used Java web-based application servers. Finally, Tomcat is embedded in many enterprise application servers that serve very high volumes of requests.
- **ActiveMQ Java Messaging Server (JMS) Broker.** Apache ActiveMQ (activemq.apache.org) is a widely used message broker that implements Java Messaging Services (JMS) [9]. It is also designed for high performance clustering, client-server, peer based communication.
- **Deployment And Configuration Engine (DAnCE).** DAnCE [10] is an implementation of the Object Management Group (www.omg.org) Deployment & Configuration (D&C) [11] specification for deploying and configuring component-based distributed systems.

These open-source applications were selected for several reasons. First, we analyzed their system execution trace with only UNITE (*i.e.*, without SETAF) and got invalid analysis because the system execution trace lacked to required properties to support analysis via UNITE. Secondly, each open-source application exhibited a different adaptation pattern, which is discussed in their respective result section. Finally, each software application used a logging facility, such as log4j [12] and ACE Logging Facilities [13]. It was therefore possible to use appenders and intercepters, respectively, to capture generated system execution traces and store them in a database for adaptation and analysis using SETAF and UNITE without making any modifications to the applications existing source code.

All experiments were conducted on a Intel core 2 Duo 2.1 GHz processor, with 3GB memory and running 32-bit Windows 7 operating system. The execution of either UNITE or SETAF, however, is not bound to a particular operating system, or execution environment.

B. Experimental Results for Applying SETAF to Apache ANT

Table IV shows the data table constructed by UNITE when analyzing ANT's system execution trace without SETAF. The end goal was to measure the average execution time of each ANT task, which is accomplished by subtracting the `finishTime` from the `startTime`. Unfortunately, Table IV constructed by UNITE will produce incorrect results because some rows are not correlated correctly. For example, first and third row have a `startTime` that is greater than the `finishTime`. This means that the task finished before it actually started, which is not the case.

TABLE IV
DATA TABLE RECONSTRUCTED BY UNITE FOR A SUBSET OF ANT'S TASKS WITHOUT ADAPTATION PATTERN SPECIFICATION.

startTime (msec)	LF1.task	finishTime (msec)	LF2.task
1500	property	860	property
1500	property	1704	property
1516	available	1511	available
1516	available	1518	available

The reason for this error is not in the generated system execution trace. Instead, the error lies in the analysis because the relations in the dataflow model used to reconstruct the dataset from the system execution trace is not unique (see Listing 3). Because of the non-unique relations in the dataflow model, the final analysis using only UNITE results in several negative values for the execution time of different ANT tasks as illustrated in Table V.

TABLE V
RESULTS FOR ANALYZING RECONSTRUCTED TABLE IN UNITE FOR ANT WITHOUT ADAPTATION SPECIFICATION.

Task	Execution Time (msec)
available	-630.333333333333
delete	0.0
macrodef	140.0
mkdir	-25.125
path	297.0
patternset	-9.76923076923077
property	-241.4
Total evaluation time (sec)	0.11994

To correct the errors in UNITE's current analysis, we defined a SETAF specification for adapting ANT's generated system execution trace for analysis. Table VI therefore highlights the dataset reconstructed by UNITE after using SETAF to apply the adaptation pattern to the reconstruction process. As shown in this table, `startTime` and `finishTime` are now correlated correctly because of the unique id added by SETAF. In this table, `startTime` is always less than `finishTime`, which is the expected result.

TABLE VI
IMPROVED TABLE RECONSTRUCTION USING SETAF AND UNITE FOR A SUBSET OF ANT'S TASKS.

LF1.uid	LF1.task	startTime	LF2.uid	LF2.task	finishTime
1	property	766	1	property	860
2	property	1500	2	property	1704
3	available	1500	3	available	1511
4	available	1516	4	available	1518

Finally, Table VII illustrates the updated final results for analyzing task execution time after using SETAF to adapt the system execution trace as UNITE analyzed it. As shown in this table all the service times for different ANT tasks have positive values, which will yield the expected (and correct) analysis results.

C. Experimental Result for Applying SETAF to Apache Tomcat

To further validate SETAF's method for adapting system execution traces for analysis via UNITE, we applied SETAF and UNITE on Apache Tomcat. To obtain a considerable amount of log messages for QoS analysis, we had to set the log level to a high value (*i.e.*, DEBUG) to produce more verbose system execution trace. This has some impact on the system performance, but the purpose of this experiment is to validate SETAF's applicability to a variety of applications—not to validate performance.

TABLE VII
FINAL RESULTS FOR ADAPTING UNITE'S ANALYSIS USING SETAF FOR A SUBSET OF ANT'S TASKS.

Task	Execution Time (msec)
available	93.6666666666667
delete	55.0
macrodef	79.0
mkdir	2.0
path	390.0
patternset	6.0
property	17.975
Total evaluation time (sec)	0.30116

When the Tomcat server starts up, it outputs the total time of the startup process. Our aim was to compare this value with the value calculated from analyzing its generated system execution traces using UNITE. The log messages related to this case study, however, do not contain any variable parts other than the timestamp of the event being capture in the system execution trace. This means that SETAF is needed to adapt the system execution trace.

After analyzing the system execution trace, we identified twelve independent events (or log formats) associated with Tomcat's startup process. Although there were no variable parts in the log formats for explicitly identifying causality in the dataflow flow, the desired causality can be defined by injecting a common id.

```
DataPoints:
  string LF1.cid;
  string LF2.cid;
  // ...
  string LF12.cid;

Relations:
   LF1.cid->LF2.cid;
   LF2.cid->LF3.cid;
   // ...
   LF11.cid->LF12.cid;

// Begin adaptation code section
On LF1:
   vars["cid"]->value ("Tomcat");

On LF2:
   vars["cid"]->value ("Tomcat");
// ...

On LF12:
   vars["cid"]->value ("Tomcat");
```

Listing 6. SETAF specification for Apache Tomcat.

Listing 6 therefore highlights a portion of the SETAF specification for Tomcat. As illustrated in this figure, a variable called *cid* is added to all the log formats to expose the hidden relation. After generating the adapter's source code from the specification UNITE was run with the adapter.

Table VIII shows the results for comparing the server

TABLE VIII
RESULTS FOR ADAPTING TOMCAT SYSTEM EXECUTION TRACE USING UNITE AND SETAF.

Method	Time (msec)
Server startup time from Tomcat instrumentation	68799.0
Server startup time from UNITE w/o SETAF	N/A
Server startup time from UNITE w/ SETAF	68802.0
Total evaluation time (sec)	9.95856

startup time calculated by UNITE without SETAF and UNITE with SETAF against the server startup time given by the server itself. As shown in this table, it was not possible to analyze the system execution trace using UNITE alone because there are no variable parts for defining causality between log formats (*i.e.* Challenge 2 in Section II). When we analyzed the same system trace using UNITE and a SETAF adaptation specification, the resulting analysis is relatively close (*i.e.*, a 0.00436% difference). The reason for the difference in time is because the instrumentation points in the Tomcat source code are not the same as the two points where the log messages are generated. More importantly, however, this experiment shows that SETAF and UNITE can be used to produce results similar to direct instrumentation. This, however, is dependent on how far a generated log message is from the real instrumentation points-of-interest.

D. Experimental Result for Applying SETAF to Apache ActiveMQ

In Apache ActiveMQ, each message broker uses a local file for persistent storage. This persistent store is updated periodically in order to prevent message lost during a system crash. This process is called *checkpointing*. When checkpointing, ActiveMQ generates a message with the content "checkpoint started". At the end of the checkpointing task, ActiveMQ generates another message with the content "checkpoint done". Because ActiveMQ checkpoints periodically, the checkpointing messages occur frequently in the generated system execution trace.

Unfortunately, when we first tried to evaluate ActiveMQ's average checkpointing time using UNITE for one scenario, we learned that average checkpointing time was -27235.333 msec. This result was clearly not correct because checkpointing time cannot be a negative number. We then realized that ActiveMQ's system execution trace cannot be analyzed as is using UNITE because ActiveMQ's dataflow model does not contain unique relations.

ActiveMQ's system execution trace, however, is similar to Apache ANT's system execution trace. This is because each log message that represents the start of checkpointing is preceded by a finish checkpointing message before another start checkpointing message occurs. Because of this fact, the same adaptation specification is used as in Listing 4 to adapt the system execution trace generated by ActiveMQ.

After auto-generating the adapter code from the SETAF specification and invoking UNITE with the SETAF adapter for ActiveMQ, we were able to evaluate that average checkpointing time was 115.917 msec the scenario discussed above that was analyzed incorrectly using UNITE alone. Distributed system testers therefore can use UNITE and SETAF to determine whether there are any performance problems with the checkpointing module of ActiveMQ without making any modifications to the existing source code to perform such analysis.

E. Experimental Result for Applying SETAF to DAnCE

The goal of analyzing DAnCE is to evaluate the amount of time it takes to deploy a set of components on a given node in the generated deployment plan. For this case study we used the BasicSP scenario provided with DAnCE. The BasicSP scenario has four different components mapped into four different nodes. After manually analyzing DAnCE's system execution trace for the BasicSP scenario, the following dataflow model was constructed for DAnCE.

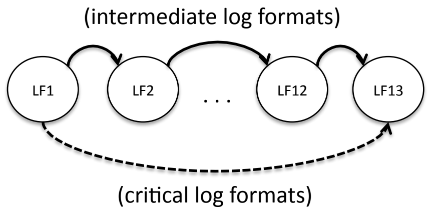

Fig. 2. Logformats associated with DAnCE

As shown in Figure 2, the dataflow model contain 13 different log formats (out of 50+ log formats) that depended on each other. To perform the necessary analysis, data in the first and last log format are sufficient. Unfortunately, because of different execution flows in DAnCE and its distributed functionality, it is not possible to use only the first and last log formats for the analysis. Instead, each intermediate log format between the first and last log format must be considered to ensure correct correlation. Unfortunately, the relation between the intermediate log formats is not unique.

Because component deployment is done according to a deployment plan it is possible to use a common id named *planid* to correlate the messages among different components and deployment plans. Moreover, another id named *nodeid* can be used to correlate messages that are generated from the same node. Listing 7 therefore presents the SETAF specification for adapting DAnCE's generated system execution traces for analysis using UNITE.

```
Variables:
  string planid;
  int lf12_count_, lf13_count_, nodeid_;

Init:
  lf12_count_ = lf13_count_ = nodeid_ = 0;

DataPoints:
  string LF1.planid; string LF2.planid;
  string LF5.planid; string LF6.planid;
  string LF9.planid; string LF11.planid;
  int LF12.nodeid; int LF13.nodeid;

Relations:
  LF1.planid->LF2.planid; LF5.planid->LF6.planid;
```

```
    LF6.planid->LF7.planid; LF8.planid->LF9.planid;
    LF9.planid->LF10.planid; LF10.planid->LF11.planid;
    LF11.planid->LF12.planid; LF12.nodeid->LF13.nodeid;

  // Begin adaptation code section
  On LF1:
    vars["planid"]->value (this->planid);
  On LF2:
    vars["planid"]->value (this->planid);
  On LF5:
    vars["planid"]->value (this->planid);
  On LF6:
    vars["planid"]->value (this->planid);
  On LF9:
    vars["planid"]->value (this->planid);
  On LF11:
    vars["planid"]->value (this->planid);
  On LF12:
    this->plan_id = SETAF::vars["planid"]->value ();
    SETAF::int_vp(vars["nodeid"])->value(this->lf12_count_);
    this->lf12_count_;
  On LF13:
    SETAF::int_vp(vars["nodeid"])->value(this->lf13_count_);
    this->lf12_count_;
```

Listing 7. SETAF specification for DAnCE.

As illustrated in Listing 7, all the log formats in the SETAF specification for DAnCE, other than `LF12` and `LF13`, use the private variable `planid` to get its adaptation value. The value of this private variable is set using `LF12` because it is the first log format SETAF processes. This is because UNITE processes the log messages in the topological order based on the dataflow model to achieve $O(n)$ runtime complexity where n is the number of log formats processed.

In order to correlate `LF12` and `LF13`, a newly added id named `nodeid` is used. The instance counts of this log format are kept in state variables `lf12_count_` and `lf13_count_`. These variables are used to populate the value of `LF12.nodeid` and `LF13.nodeid`. This allows us to differentiate the similar instances of the same log format. The scenario we tested with DAnCE has nodes named `EC`, `BMDevice`, `BMClosedED` and `BMDisplay` where each contains a set of component instances.

TABLE IX
RESULTS FOR ADAPTING DANCE'S SYSTEM EXECUTION TRACE USING SETAF TO MEASURE DEPLOYMENT TIME (DT).

Node	DT w/o SETAF	DT w/ SETAF
EC	N/A	30.0
BMDevice	N/A	30.0
BMClosedED	N/A	30.0
BMDisplay	N/A	31.0
Total evaluation time (sec)	N/A	0.35542

Table IX illustrates results for analyzing deployment time for each node after adapting DAnCE's generated system execution trace while undergoing analysis using SETAF and UNITE. As shown in this table, we were not able to analyze DAnCE's system execution trace using only UNITE because some of the log formats were lacking variables parts to define causalities (*i.e.* Challenge 2 in Section II) and some newly added log format variables required analyzing other log messages to populate their corresponding value (*i.e.* Challenge 3 in Section II). When we used both UNITE and SETAF to analyze DAnCE's system execution trace, we learned that all four nodes take approximately equal time to deploy. More importantly, however, these results show that with careful analysis of the generated system execution trace, SETAF and UNITE can be used to analyze such complex dataflow models as found in DAnCE without modifying the existing source code.

Final discussion. The four case studies above showcase three different adaptation patterns. The ANT case study and ActiveMQ case study are similar in that a unique id is added to the two log formats. In the DAnCE case study, the new log format variables were populated using the values extracted from previously found log format variable values after analyzing the system execution trace. In the Tomcat case study, a common id is added to log formats. This, therefore, showcases the flexibility and extendibility of SETAF's approach to support adaptation of system execution traces to support QoS analysis.

V. Related Work

Fischer et al. [14] describe a technique to track system evolution by analyzing the system execution traces. Their method locates execution patterns in the traces to determine how the software has evolved between different revisions (or versions). Their method also uses relational data model to store the traces and assign unique ids for analysis purposes. SETAF differs from their work, because it analyzes the QoS properties of the system. Further their method requires the system execution traces to have a particular format, similar to UNITE. SETAF, however, is designed to adapt existing system execution traces so it contains properties expected by UNITE (*i.e.*, the QoS analysis tool). It is therefore believed that SETAF's method could be used to adapt system execution traces that do not contain the properties required by Fischer's method for analyzing system evolution via system execution traces.

Safyallah et al. [15] describe a method of mining system execution traces to find out common functionality associated with feature-specific task scenarios (*i.e.*, core functions that implement software features) to discover relationships between different feature specific tasks in inter/intra modules (*i.e.*, inside a single software component and between different software components). Their approach does not need any particular format in the system execution traces. Moreover, their main focus is to evaluate the functional aspects of the system using system execution traces. In contrast, SETAF's approach is to support validation of software system QoS properties, such as end-to-end response time, service time and throughput via analyzing system execution traces.

Voigt et al. [16] present a trace visualization technique for analyzing method calls and object access. Their main purpose is to understand the large execution traces as a sequence of object activities. Similar to SETAF, they are also trying to find out relationships in the traces, which are mapped to relationships as activities between objects. SETAF differs from this work, because its is trying to do analysis at a higher level than object activities. SETAF tries to analyze the traces in an architecture and implementation independent manner.

Moreover, SETAF does not have the visualization aspects. When SETAF is combined with UNITE, however, it can be used to visualize the data trends of the corresponding QoS property under analysis.

VI. Concluding Remarks

System execution traces are information rich resources that can be used to understand system properties, such as its quality-of-service (QoS). This, however, is only possible if the system execution traces contain the required properties to support such analysis. This paper presented the *System Execution Trace Adaptation Framework (SETAF)*, which is a technique and a framework that adapts system execution traces for QoS validation. SETAF operates by applying adaptation patterns to existing system execution traces to ensure correct analysis. As shown in the results from applying SETAF to several open-source software systems, it is possible to perform such adaptation without modifying the original source code.

Based on experience gained from applying SETAF to these open-source software applications and systems, the following is a list of lessons learned and future research directions:

- **Automatically identifying relations among existing log formats for adaptation.** Distributed systems can easily generate a system execution traces that are quite large. When the generated system execution trace is very large, it can be *hard* for distributed system testers to manually identify adaptation patterns—especially without adequate domain knowledge. Future work therefore includes applying existing data mining techniques to assist in locating adaptation patterns—thereby easing complexities associated with the analysis.
- **Automatically identifying the dataflow model from a system execution trace.** In relation to the previous lesson learned, when the generated system execution trace is very large, it is *hard* to identify a valid dataflow model—irrespective of the dataflow model having properties required for QoS analysis. Future work therefore includes applying existing data mining techniques to assist in identifying the dataflow model—thereby easing complexities associated with the analysis.
- **Instrumentation points have direct impact on result accuracy.** When analyzing QoS properties using system execution traces it is important that the log messages are inserted in the correct location in the source code—especially when analyzing time-related QoS properties as shown with the Tomcat results (see Section IV-C). More importantly, it is critical to understand execution semantics to prevent inaccurate results.

SETAF and UNITE are integrated into the CUTS system execution modeling tool. They are freely available in open-source format for download from the following location: http://cuts.cs.iupui.edu.

References

[1] F. Chang and J. Ren, "Validating System Properties Exhibited in Execution Traces," in *Proceeding of the* 22^{nd} *IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 517–520.

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

[3] J. H. Hill, H. A. Turner, J. R. Edmondson, and D. C. Schmidt, "Unit Testing Non-functional Concerns of Component-based Distributed Systems," in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*, Denver, Colorado, apr 2009.

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

[5] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2008, pp. 117–126.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[7] *Java Servlet Specification* , Version 3.0 ed., SUN, Dec. 2009.

[8] *Java Server Pages Specification* , Version 2.1 ed., SUN, May 2006.

[9] SUN, "Java Messaging Service Specification," java.sun.com/products/jms/, 2002.

[10] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale, "DAnCE: A QoS-enabled Component Deployment and Configuration Engine," in *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, Grenoble, France, Nov. 2005, pp. 67–82.

[11] *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 ed., Object Management Group, Jul. 2003.

[12] Apache, "Logging Services," http://logging.apache.org/index.html.

[13] D. C. Schmidt, "The ADAPTIVE Communication Environment (ACE)," www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

[14] M. Fischer, J. Oberleitner, H. Gall, and T. Gschwind, "System evolution tracking through execution trace analysis," in *13th International Workshop on Program Comprehension, 2005. IWPC 2005.*, May 2005.

[15] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 2006, pp. 84–88.

[16] S. Voigt, J. Bohnet, and J. Dollner, "Object aware execution trace exploration," in *IEEE International Conference on Software Maintenance, 2009. ICSM 2009.*, September 2009.