Adapting System Execution Traces to Support Analysis of Software System Performance Properties

Manjula Peiris*, James H. Hill

Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA

Abstract

UNITE is a method and tool that analyzes software system performance properties, *e.g.*, end-to-end response time, throughput, and service time, via system execution traces. UNITE, however, assumes that a system execution trace contains properties (*e.g.*, identifiable keywords, unique message instances, and enough variation among the same event types) to support performance analysis. With proper planning, it is possible to ensure that properties required to support such analysis are incorporated in the generated system execution trace. It, however, is not safe to assume this to be the case with many existing software systems.

This article therefore presents a method and a tool called the *System Execution Trace Adaptation Framework (SETAF)*, which is built atop of UNITE and adapts system execution traces to support performance analysis of software systems. It also presents examples and results of applying SETAF to different open-source projects. The results show that SETAF enables proper performance analysis via system execution traces without requiring developers to make modifications to the originating software system's source code, which can be a expensive and time-consuming task.

Keywords: SETAF, performance analysis, system execution traces, adaptation

1. Introduction

Dynamic software analysis (Bell 1999) is the process of analyzing a software system by executing it. Analysis of system execution traces is one technique used in dynamic software analysis (Safyallah and Sartipi 2006; Breu and Krinke

*Corresponding author
Email addresses: tmpeiris@cs.iupui.edu (Manjula Peiris), hillj@cs.iupui.edu (James H. Hill)

2004; Cornelissen et al. 2009). There are several methods for generating system execution traces, such as:

1. Compiling the original source code with an instrumentation code and executing the combined executable (Geimer et al. 2010; Wolf and Mohr 2003; Wylie et al. 2007);
2. Collecting log messages (*i.e.*, traces generated from log statements in the source code) during system execution (Erkki Salonen 2012; Nagaraj et al. 2012; Han et al. 2012; Hill et al. 2009; Hill 2010); and
3. Registering for certain events of the target system and generating messages whenever the events occur (Mos et al. 2001; Mania et al. 2002; Parsons et al. 2006).

The first method is intrusive because it involves modifying source code for analytical purposes. The second and third methods are considered less intrusive because it requires minimum changes to the target system.

System execution traces have been used on functional aspects of the system, such as detecting system failures (Xu et al. 2008), operational profiling (Nagappan et al. 2009), and website usage patterns based on user sessions (Silverstein et al. 1999). Some Other research efforts (Yin et al. 2012; Wolf and Mohr 2003) are focusing on using system execution traces to do performance analysis. One benefit of using system execution traces to analyze software system performance properties is the system execution trace provides a comprehensive view of the system's behavior and state throughout the system's execution lifetime. This is opposed to a single snapshot of the system at a given point in time, such as a global snapshot (Singhal and Shivaratri 1994). Likewise, they provide system testers with a rich set of data for analyzing data trends associated with a given performance property, *i.e.*, how a given performance property changes with respect to time.

Existing approaches for using system execution traces to analyze performance properties rely on intrusive methods to collect traces (Geimer et al. 2010; Wolf and Mohr 2003). Because these methods require access to the system's original source code, it is *hard* to apply these approaches when the source code is not available. Other approaches for validating performance properties via system execution traces are tightly coupled to system's architecture and technology (Mos et al. 2001; Mania et al. 2002; Parsons et al. 2006). Finally, the approaches that are not implementation dependent require system execution traces to be generated in a certain format (Erkki Salonen 2012; Nagaraj et al. 2012). Moreover, such approaches are not trying to utilize system log messages, but rather enforce the system implementers to use the provided logging mechanisms or convert the system logs to an intermediate format. Unfortunately, the limitations discussed above make it hard to generalize the existing approaches for different kinds of systems, and their generated system execution trace.

By system execution traces, we mean a collection of log messages with a timestamp that describe certain events in the system, and are collected without making any modification to the existing source code or binary. Our current and previous research efforts have focused on using system execution traces

to analyze software system performance properties without any modification to the original source code, or converting to an intermediate format (Hill et al. 2009).We have realized our technique in a tool called *Understanding Non-functional Intensions via Testing and Experimentation (UNITE)* (Hill et al. 2009). UNITE accomplishes this feat by using dataflow models that describe causality relationships between event types—not event instances—in the system. This allows UNITE to operate at a higher level of abstraction that remains constant regardless of how the underlying software system is designed, implemented, and deployed (*i.e.*, the mapping of software components to hardware components). More details on UNITE are provided in Section 2.

Although it is possible to analyze performance properties via system execution traces using tools like UNITE (without any restriction on log message format), it is assumed that system execution traces contain several properties, *e.g.*, identifiable keywords, unique message instances, enough variations among the same event types to support performance 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 the properties required to support performance analysis via UNITE.

This article therefore presents a method for adapting system execution traces and their dataflow models to contain properties required to analyze software system performance properties. The main contributions of this article, which extends prior work presented in (Peiris and Hill 2012), are as follows:

- It presents the *System Execution Trace Adaptation Framework (SETAF)*—a framework for adapting system execution traces and dataflow models to contain the properties required to support performance analysis. It is also the first attempt, to the best of the author's knowledge, of a method and a tool for adapting system execution traces "as-is" for performance analysis;
- It discusses different design alternatives—including their advantages and disadvantages—for adapting system execution traces to support performance analysis;
- It showcases how SETAF can be applied to system execution traces generated by different software systems; and
- Apart from the above contributions as an increment to (Peiris and Hill 2012), it compares the execution performance of two different system execution trace adaptation techniques for a given adaptation specification (*i.e.*, compiled versus interpreted adaptation).

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 originating source code. Likewise, experimental results

showcase that an interpreted version of the adaptation specification shows no degradation in performance when compared to its compiled version.

Article organization. The remainder of this article is organized as follows: Section 2 provides a detailed overview of UNITE and its current limitations; Section 3 presents the design and functionality of SETAF; Section 4 discusses the results of applying SETAF to several open source projects, and compares the performance of two different SETAF adaptation techniques; Section 5 compares SETAF to other related works; and Section 6 provides concluding remarks and future research directions.

2. A Brief Overview of UNITE

UNITE is a method and tool for analyzing system execution traces and validating software system performance properties. UNITE's analytical techniques are not tightly coupled to (1) system implementation *i.e.*, what technologies are used to implement the system, (2) system composition, *i.e.*, what components communicate with each other, and (3) deployment, *i.e.*, where components are located. This is opposed to processing system execution traces using simple scripts where those scripts are typically hard-coded for a specific use case, system, and/or problem. 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.

2.1. Understanding UNITE

Table 1 presents an example system execution trace from a software system. As shown in this table, the system execution trace has log messages that correspond to sending/receiving events between components in a software system. The log messages in this example contain time of the event, event id, and name of the component where the event occurred.

Time of Day	**Hostname**	**Message**
2012-01-25 05:15:55	node1	Config: sent event 1 at 120394455
2012-01-25 05:15:55	node2	Planner: sent event 2 at 120394465
2012-01-25 05:15:55	node2	Planner: received event 2 at 120394476
2012-01-25 05:15:55	node1	Config: received event 1 at 120394480
2012-01-25 05:15:55	node3	Effector: sent event 3 at 120394488
2012-01-25 05:15:55	node3	Effector: received event 3 at 120394502

Table 1: An example system execution trace displayed in table format as if being stored for offline analysis in a database.

Software system testers use UNITE to analyze performance properties from the system execution trace in Table 1 by first identifying what log messages to extract from the execution trace. These log messages should contain metrics of interest that support desired performance analysis. Once the log messages are identified, software system testers convert the common log messages into

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. The variable portions are those that change between different log messages.

Given the system execution trace in Table 1, Listing 1 shows the log formats that represents the different log messages in the trace. 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.

```
1  LF1: {STRING cmpid} sent event {INT eventid} at {INT sent}
2  LF2: {STRING cmpid} received event {INT eventid} at {INT recv}
```

Listing 1: Log formats for analyzing system execution trace presented in Table 1.

A log format LF_i can have zero or more variables. We define the set of variables of log format LF_i as V_i. Similarly, the set of variables of log format LF_j is V_j. A *causal relation* $CR_{i,j}$ between two log formats LF_i and LF_j is denoted as $LF_i \rightarrow LF_j$ where LF_i is the cause log format and LF_j is the effect log format. This kind of a causal relation is also called a *log format relation*. A causal relation $CR_{i,j}$ can have zero or more variable relations. A *variable relation* $VR_{C,E}$ of a causal relation $CR_{i,j}$ is defined as $v_C = v_E$, where $v_C \in V_i$ and $v_E \in V_j$. A system execution trace can have many log formats, many causal relations between the log formats, and many variable relations for each causal relation.

For the purpose of performance analysis, system testers can use subsets of the log formats, causal relations and variable relations, which we call a *dataflow model*. We formally define a dataflow model $DM = (LF, CR, VR)$ as:

- A set LF of log formats where each log format represents a set of log messages useful for analyzing a performance property;
- A set CR of causal relations that specify order of occurrence and causality among the log formats LF; and
- A set VR of variable relations attached to causal relations CR.

Like dataflow models in program analysis (Allen and Cocke 1976) where they relate variables across different source lines, dataflow models in UNITE relate log format variables across different log messages (or application contexts). The dataflow model then 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 (Singhal and Shivaratri 1994). This is because the relations between the log formats preserve causality. For the system execution trace in Table 1, Listing 2 illustrates the dataflow model.

```
1  Log Formats:
2   LF1: {STRING cmpid} sent event {INT evid} at {INT sent}
3   LF2: {STRING cmpid} received event {INT evid} at {INT recv}
```

```
4
5  Log Format Relations:
6   LF1 → LF2
7
8  Variable Relations:
9   LF1.cmpid = LF2.cmpid
10  LF1.evid = LF2.evid
```

Listing 2: Dataflow model for analyzing system execution trace presented in Table 1.

The dataflow model illustrated in Listing 2 is a higher level abstraction of the system execution trace being analyzed. Using this dataflow model and the system execution trace, UNITE creates a variable correlation table—based on variables defined in the dataflow model. A variable correlation table is a set of tuples $(d_1, d_2, .., d_i, .., d_n)$ where each tuple $d_i (i \leq n)$ contains instance values for all the variables of log formats defined in the dataflow model. The correlation of these values (*i.e.*, the values occur together in a single tuple) is defined by the variable relations in the dataflow model.

Our previous work on UNITE (Hill et al. 2009) illustrates an algorithm based on topological sorting directed acyclic graphs and constructing variable correlation table for a given system execution trace and a dataflow model. Because each tuple in the variable correlation table is an instance of a variable relation in the dataflow model, it is important that the log messages are processed in the correct order. This ordering is defined by the reverse topological ordering of the log formats in the dataflow model. The dataflow model is a directed acyclic graph where each log format is a node and for each log format relation there is an edge from cause log format to effect log format. For a causal relation if we can find a log message instance representing an effect log format, then we should be able to find the corresponding cause log message instance from the same system execution trace.

The reverse, however, is not always true. For example, there can exist a "sent" event without a "received" event. On the other hand, if there is a "received" event, then the corresponding "sent" event must exist in the same system execution trace. Once the log formats in the dataflow model are topologically ordered, the source node (cause log format) comes first in the list. Because UNITE only considers cause log message instances that have the corresponding effect log message instance, UNITE has to process effect log message instances first. UNITE therefore processes the log messages in the reverse topological order.

For example, for any $LF_i \rightarrow LF_j$ log format relation, UNITE first processes the log messages that correspond to LF_j, and then log messages that correspond to LF_i. While processing in this order, UNITE populates the variable correlation table based on the values of the variables of each log message instance. For example, Table 2 shows the variable correlation table for the system execution trace in Table 1 and dataflow model in Listing 2.

The variable correlation table shown in Table 2 enables software system testers to analyze performance properties. For example, Listing 3 highlights the expression for evaluating average event round trip time.

LF1.cmpid	LF1.evid	LF1.sent	LF2.cmpid	LF2.evid	LF2.recv
Config	1	120394455	Config	1	120394480
Planner	2	120394465	Planner	2	120394476
Effector	3	120394488	Effector	3	120394502

Table 2: Variable correlation table for the system execution trace shown in Table 1.

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

Listing 3: Expression for analyzing round trip time using UNITE.

Based on this expression, UNITE can generate SQL queries that can aggregate performance results captured from the variable correlation table. Likewise, if the aggregation function (*i.e.*, `AVG`) is removed from the expression, then UNITE will present the data trend for the performance property undergoing analysis. Lastly, UNITE provides facilities to group aggregated results—similar to grouping in SQL.

2.2. Current Limitations of UNITE

Although UNITE enables analysis of software system performance properties using system execution traces, UNITE's methodology has the following limitations:

- **Challenge 1: Correlating log formats that have non-unique instances.** As mentioned above the variable correlation table is a set of tuples. Each tuple is a set of values for log format variables. Some of these values represent a time stamp for a particular event (*e.g.*, LF1.sent, LF2.recv). Let us define the subset of a tuple that does not represent time as F. UNITE assumes that the set F of any tuple in the variable correlation table to be unique (*i.e.*, any instance of a particular log format is different from any other instance of the same log format apart from the timestamps). We call this *uniqueness* among the log messages.

  As shown in Table 1 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 situation occurs, 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.

Time of Day	Hostname	Message
2011-02-25 12:00:55	node1	Started doing task A at 12.00
2011-02-25 12:01:55	node1	Finished doing task A at 12.01
2011-02-25 12:02:55	node1	Started doing task A at 12.02
2011-02-25 12:03:55	node1	Finished doing task A at 12.03

Table 3: Example system execution trace that does not contain unique ids.

For example, Table 3 illustrates an example system execution trace where the different instances of the same log format are similar. Only the variable parts related to time change in different instances. It is therefore *hard* to know what start/finish messages are associated with each other without human intervention. Moreover, when an example similar to the one presented in Table 3 is analyzed by UNITE, it will yield incorrect results (see Section 4.2 for supporting 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 1, 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 contain *hidden relations*.

Time of Day	**Hostname**	**Message**
2011-02-25 10:00:55	node1	Initializing the system at 10.00
2011-02-25 10:10:55	node1	Start Monitoring components at 10.10
2011-02-25 10:11:55	node1	Finish Monitoring components at 10.11
2011-02-25 10:40:55	node1	Shutting down the system at 10.40

Table 4: Example system execution trace that contain hidden relations.

For example, consider the system execution trace in Table 4. 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 4 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 3 and Table 4, however, requires adding new log format variables to 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 software systems. Although system execution traces vary from system to system, it is possible to use a general-purpose approach for

adapting them so that it will help to correctly correlate events in the system. This kind of correct correlation will be useful to support performance analysis of the system. The next section therefore explains how SETAF addresses the challenges outlined above to enable performance analysis using system execution traces for software systems that would otherwise not be able to support such analysis.

3. 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.

3.1. Design Approaches for Adapting System Execution Traces

Before discussing the details of SETAF, it is necessary to understand different approaches for adapting system execution traces and dataflow models to support performance analysis—in particular with UNITE. The following therefore is a list of approaches for realizing the adaptation:

- **Approach 1: Change system execution traces directly.** In come cases, system execution traces may not contain certain properties that enable performance analysis. The execution semantics of those systems, however, can be used to define dataflow models as defined in Section 2. If the system's source code is available, then the source code can be changed so that the system execution trace reflects the execution semantics. For example, the source code that generates the system execution trace can be changed such that each log format has variables for capturing unique relations.

  The advantage of this approach is that UNITE—as is—can directly analyze generated system execution traces. This approach, however, has several disadvantages. First this approach requires software system testers to have access to the source code so they can make the necessary updates. Moreover, it requires software system testers to be familiar with the source code—its implementation—to make the necessary updates. Secondly, this approach is not practical because updating the source code accordingly can be a costly, error prone, and time consuming task—especially when dealing with a large code base. Finally, the actual source code should not be changed just to analyze performance properties because such changes may impact existing functional and performance properties 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.

  The advantage in this approach is that the source code of the actual software system does not need to change. 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 performance 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 reside in an external specification. The external specification is then loaded by UNITE when analyzing the corresponding system execution trace. This approach allows software system testers to write their own adaptation specification according to the system domain without modifying the software system's existing source code.

  The disadvantage of this approach is that software system testers must be aware of the dataflow model's limitations. Software system testers also need to identify the new log format variables, the relations that must be added to the existing dataflow model so it can be adapted correctly.

Based on the advantages and disadvantages of each approach discussed above, Approach 3 was selected as the approach for adapting system execution traces for performance analysis. 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.

When using Approach 3 (above), software system testers first have to identify the adaptation pattern using domain knowledge about the system execution trace. This adaptation pattern then needs to be expressed in a manner that UNITE can understand. One possible way of doing this is to implement the adapter(s) in the same programming language as UNITE, which is C++. Although this is possible, software system testers will be required to possess some domain knowledge about underlying architecture and functionality of UNITE. We therefore designed a domain-specific language for expressing the adaptation pattern, which does not require software system testers to know the internals of UNITE. Section 3.2 describes this domain-specific language in detail.

3.2. Defining the Adaptation Specification

As discussed in the previous section, the approach of using external adapters was selected for adapting system execution traces for performance analysis. Software 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 to support performance analysis via the system execution trace. Each adaptation specification contains the following details:

- **Variables.** The variables are private data points that assist with adapting the corresponding system execution trace. The variables are visible only to the adaptation pattern, and not visible to UNITE—thereby helping to address Challenge 3 introduced in Section 2.2. Moreover these variables keep the state of the current adaptation throughout the system execution trace analysis.

- **Initialization.** The initial values for the variables defined above are specified in this section.

- **Reset.** The state variables defined above may need to be reset at the start of processing each log format. This section contains the values for such resetting.

- **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 2.2.

- **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 2.2.

- **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 2.2.

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 3. We selected 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 4.

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 a build process. A task finish event is the effect of a task start event. Using this domain knowledge of the system execution trace, Listing 4 illustrates the dataflow model for analyzing the execution time of each task in Apache ANT.

```
Log Formats:
  LF1:Started doing task {STRING taskname} at {INT startTime}
  LF2: Finished doing task {STRING taskname} at {INT finishTime}

Relations:
  LF1.taskname = LF2.taskname
```

Listing 4: 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.*, different only in the time stamp) in the system execution trace. When UNITE is processing the system execution traces using the dataflow model shown in Listing 4, it first identifies all the log message instances of type `LF2`. Then for each message of that type, UNITE tries to find the corresponding `LF1` message instance (*i.e.*, UNITE is trying to correlate the finish event of an ANT Task with the start event of the same task). As shown in the Listing 4, the only possible way to do this is using the `taskname`. Because `taskname` is not always different among different message instances, UNITE cannot do this correlation correctly.

This behavior in UNITE is similar to Challenge 1 described in Section 2. Although ANT's system execution trace has this problem, a log message representing the start of a task is *always* preceded by a log message representing completion of the corresponding task. The system tester knows this is the execution semantics of ANT, but it is not completely captured in the system execution trace. This observation is therefore used to write a SETAF specification that adapts ANT's dataflow model accordingly. Listing 5 highlights the adaptation pattern—written as a SETAF specification—to ensure correct analysis of ANT's system execution trace.

```
Variables:
  int id_;

Init:
  id_ = 0;

Reset:
  id_ = 0;

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

Relations:
  LF1.uid -> LF2.uid;

// Begin adaptation code section
On LF1:
  id_ = id_ + 1;
  [uid] = id_;

On LF2:
  id_ = id_ + 1;
  [uid] = id_;
```

Listing 5: Adaptation pattern specification for Apache ANT in SETAF.

As illustrated in this listing, first software system testers define variables needed to adapt the system execution trace. This information is captured in the

section labeled *Variables* of the SETAF specification. In this case the variable named *id_* maintains the state of the adaptation. As shown in section labeled *Init* and section labeled *Reset* the value of this variable is initialized to 0 at the start of the adaptation and reset to 0 each time a new log format is processed. Software 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, software system testers define new relations that should be added to the dataflow model. As illustrated in Listing 5, the left side of the arrow represents the cause variable; whereas, the right side of the arrow represents the 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. This task is completed by stating how the adapter transforms the system execution trace for each log format that needs adaptation. As shown in Listing 5, 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 that the next occurrence of `LF1` is differentiated from the previous occurrence of `LF1`, as well as `LF2`. Finally, the variable inside the brackets [ ] represents log format variables in UNITE. Writing the variable inside brackets is used to differentiate the adapter state variables from UNITE's log format variables.

Integrating SETAF with UNITE. We extended UNITE to provide a configuration option for specifying the location of the adaptation specification, and a standard interface to support the functionality of the adaptation specification described above. The unified interface of UNITE defines three main methods—`update_log_format(), update_relations(), update_values()`. The implementation of these functions are defined in the adaptation specification. When there is an adapter specification provided with a dataflow model, UNITE calls the three methods above as follows.

1. **update_log_format.** This method is called when UNITE is processing log formats in the dataflow model, and SETAF needs to add data points defined in the adaptation specification to the dataflow model.
2. **update_relations.** This method is called when UNITE is processing the log format relations in the dataflow model and SETAF needs to add log format relations defined in the adaptation specification to the dataflow model.
3. **update_values.** This method is called when UNITE needs to populate the variable correlation table. Moreover, this method is only called for the columns (*i.e.*, log format variables) in the variable correlation table that are added from the adaptation specification because the system execution trace does not have values to populate the newly added columns.

3.3. Compiled versus Interpreted External Adapters in SETAF

There are two possible ways to use this adaptation specification with SETAF: *compiled adapter* and *interpreted adapter*. As shown in Figure 1 when using the compiled adapter technique, SETAF generates C++ source code using the adaptation specification. Software system testers then compile the auto-generated code into an external module. During the system execution trace analysis, UNITE loads the external module and invokes required functionality for the adaptation from the external module.

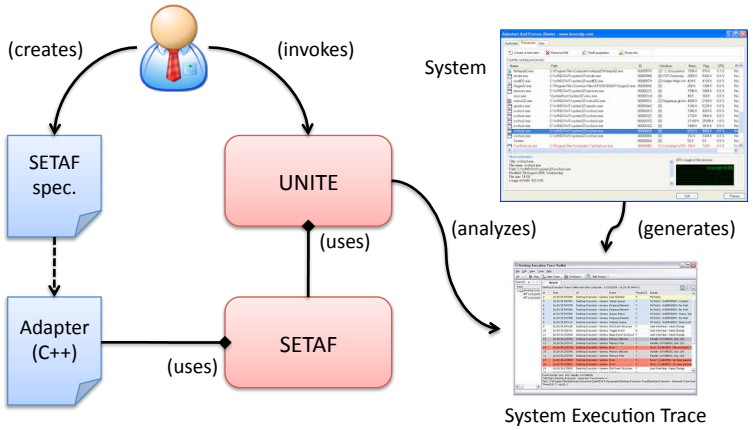

Figure 1: Overall analysis process with SETAF compiled adapter technique.

Listing 6 showcases the source code auto-generated for ANT's adapter based on the SETAF specification in Listing 5. 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()` method. More specifically, these data points are used to create new log format variables.

Similarly, the `update_relations()` method uses the relations specified in the *Relations* section of the specification. This method is therefore responsible for creating new relations among log formats with respect to the new log format variables. The `update_values()` method 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 as defined in the dataflow model.

The identifier `SETAF::int32_vp ()` represents a log format variable casting operator. It is needed because all the variables types in UNITE are derived from a common variable type. This casting operator allows the system 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. Lastly, UNITE uses SETAF to adapt its dataflow model to support the analysis of the system execution trace using the compiled version of the specified adapter source code.

```
1  class Ant_Adapter : public CUTS_Log_Format_Adapter {
2  public:
3    void init (void) { this->id_ = 0; }
4    void reset (void) { this->id_ = 0; }
5    void close (void) { delete this; }
6
7    void update_log_format(CUTS_Log_Format * lfmt) {
8      const string & name = lfmt->name ();
9
10     if (name == "LF1")
11       lfmt->add_variable ("uid", "int");
12     else if (name == "LF2")
13       lfmt->add_variable ("uid", "int");
14   }
15
16   void update_relations(CUTS_Log_Format * lfmt) {
17     const string & name = lfmt->name ();
18
19     if (name == "LF1")
20       lfmt->add_relation ("LF2", "uid", "uid");
21   }
22
23   void update_values(Variable_Table & vars, CUTS_Log_Format * lfmt) {
24     const ACE_CString & name = lfmt->name ();
25
26     if (name == "LF1") {
27       ++this->id_;
28       SETAF::int32_vp (vars["uid"])->value (this->id_);
29     }
30     else if (name == "LF2") {
31       ++this->id_;
32       SETAF::int32_vp (vars["uid"])->value (this->id_);
33     }
34   }
35
36 private:
37   int id_;
38 };
```

Listing 6: Auto-generated source code for Apache ANT adapter.

The interpreted adapter technique removes the extra overhead of code generation and compilation. As shown in the Figure 2, software system testers just need to express the adaptation pattern as a specification. During analysis time of the system execution trace, UNITE loads the adaptation specification and SETAF builds an in memory object model of the adapter.

For the interpreted adapter technique, the adaptation specification is mapped into a `SETAF_Interpreter` object type. The data points and relations are stored in two list objects where each entry in the list corresponds to a data point and relation, respectively. The methods `update_log_format()` , `update_relations()` and `update_values()` are methods on the `SETAF_Interpreter` object type. The first two methods process the data point and relation list objects, respectively, and update the dataflow model accordingly.

The interpreted adapter technique also defines two classes: `SETAF_Variable` and `SETAF_Command`. `SETAF_Variable` represents different kinds of variables in the adaptation specification, and `SETAF_Command` represents each statement in the adaptation code section of the adaptation specification. Figure 3 illustrates the `SETAF_Variable` and `SETAF_Command` classes, their subclasses, and the relationship between the two class hierarchies.

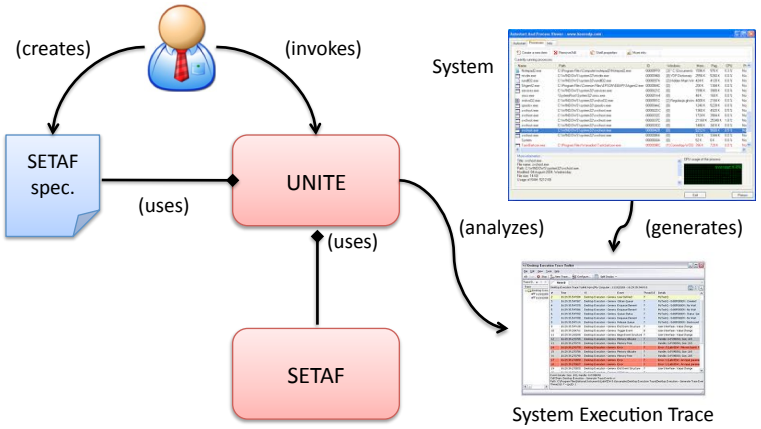

Figure 2: Overall analysis process with SETAF interpreted adapter technique.

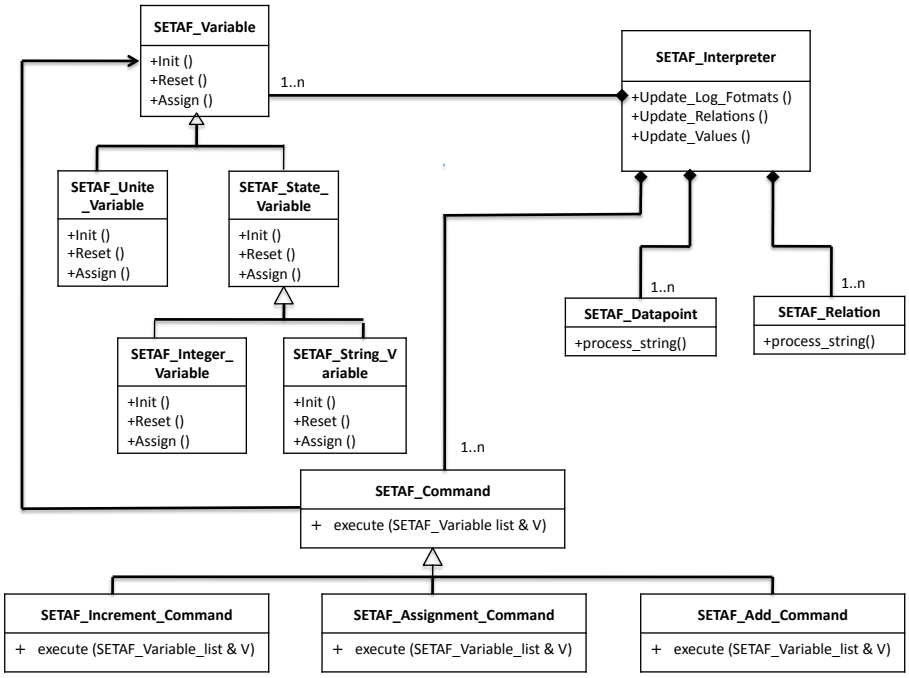

Figure 3: SETAF_Variable and SETAF_Command class hierarchies.

As shown in the figure, the variables defined in the *Variable* section of the adaptation specification are represented using `SETAF_State_Variable` object type. Likewise, UNITE's log format variables (*i.e.*, variable inside "[ ]") are represented using `SETAF_Unite_Variable` object type. Figure 3 also shows that SETAF's interpreter adapter technique currently implements three types of statements: assignment, increment, and addition. The three statements, which are an implementation of the Command pattern (Gamma et al. 1997), are represented using `SETAF_Assignment_Command`, `SETAF_Increment_Command`, and `SETAF_Addition_Command`, respectively. SETAF only implements these three statements because they are sufficient to adapt dataflow models of system execution traces we have currently used with SETAF. Further these three statements are the building blocks for most of the complex adaptation patterns. Moreover if the adaptation requires a new statement, then it can be easily implemented using the Command pattern.

The actual functionality of the command is implemented in a method named `execute()`. As described before, the adaptation code is segmented based on the log formats. Each log format that appears in the adaptation code is therefore mapped to a `SETAF_Command` object. When UNITE needs SETAF to apply adaptation to the system execution trace, it invokes `update_values()` for the corresponding log format. The SETAF interpreted adapter then locates the corresponding `SETAF_Command` object and executes it.

The interpreted adapter technique is easier to use, because the overhead associated with compiling the adapter, such as obtaining UNITE libraries and setting up a development environment, are not present in the interpreted adapter. One therefore may question the use of the compiled interpreter technique. We developed the compiled adapter technique because our initial intuition was that the compiled technique would have better performance. This is because UNITE calls the functionality for adaptation from a compiled module compared to parsing the specification file and building an object model in the interpreted adapter. Section 4.6 shows a performance comparison of the two techniques to evaluate whether our intuition is correct.

4. Results of Applying SETAF to Open-Source Projects

This section presents results from applying SETAF to several open-source projects that generate system execution traces and do not have the properties required for performance analysis via UNITE. It also shows a performance comparison of the two techniques—compiled versus interpreted adapter. Finally, the adaptation specification used for either technique (*i.e.*, compiles vs. interpreted) is the same. The performance analysis results from applying UNITE and SETAF to different open source software systems is therefore the same for both the cases.

4.1. Experimental Setup

To determine applicability of SETAF's technique, we applied SETAF to the following open-source projects:

- **Apache ANT.** Apache ANT, which was previously introduced in Section 3.2, 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 (2009) and JavaServer Pages Sun Micro Systems (2006) 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) (2002). Apache ActiveMQ supports implementation of enterprise integration patterns such as publisher-subscriber. It is also integrated into a variety of Enterprise Service Bus (ESB) Chappell (2004) middlewares in order to support message mediation. It is also designed for high performance clustering, client-server and peer based communication.

- **Deployment And Configuration Engine (DAnCE)**. DAnCE (Deng et al. 2005) is an implementation of the Object Management Group (`www.omg.org`) Deployment & Configuration (D&C) (2003) specification for deploying and configuring component-based distributed systems.

These open-source projects were selected for several reasons. First, we analyzed their system execution trace with only UNITE (*i.e.*, without SETAF) and produced invalid results because the system execution trace lacked the required properties to support performance analysis via UNITE. Secondly, each open-source project exhibited a different adaptation pattern, which is discussed in their respective result section. Finally, each software application used a logging facility, such as log4j (`http://logging.apache.org/index.html`) and ACE Logging Facilities (Schmidt 1997). 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 without making any modifications to the 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 as long as the operating system supports the Adaptive Communication Environment (ACE) (Schmidt 1993), Boost (`www.boost.org`), and SQLite (`www.sqlite.org`) middelware.

4.2. Experimental Results for Applying SETAF to Apache ANT

Table 5 shows the correlation 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 5 constructed by

UNITE will produce incorrect results because some rows are not correlated correctly. For example, the first and third rows have a `startTime` that is greater than the `finishTime`. This means that the task finished before it actually started, which is not the case.

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

Table 5: Data table reconstructed by UNITE for a subset of ANT's tasks without adaptation pattern specification.

The reason for this error lies 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 are not unique (see Listing 4). More specifically, UNITE processes the log formats in topological order of the corresponding directed acyclic graph of the data flow model. UNITE therefore first populates the `finishTime` column. Then it uses a *SQL UPDATE* query to update the corresponding data value of the `startTime` column.

In this case, UNITE can only do the correlation using the relation *LF1.task = LF2.task*. This cause UNITE to update multiple rows (as the relation is not unique) and finally ends up with the latest value of the *startTime* for a particular *task*. For example for *task property*, UNITE updates the `startTime` with 1500 by replacing all the previously updated values. Because of the non-unique relations in the dataflow model, the final analysis using only UNITE results in several negative values for the average execution time of different ANT tasks as illustrated in Table 6.

Task	**Average 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

Table 6: Results for analyzing reconstructed table in UNITE for ANT without adaptation specification.

To correct the errors in UNITE's current analysis, we defined a SETAF specification as described in Listing 5 for adapting ANT's generated system execution trace and the corresponding dataflow model. Table 7 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.

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

Table 7: Improved table reconstruction using SETAF and UNITE for a subset of ANT's tasks.

Finally, Table 8 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 produce the expected (and correct) analysis results.

Task	Average 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 of compiler technique (sec)	0.210
Total evaluation time of interpreter technique (sec)	0.220

Table 8: Final results for adapting UNITE's analysis using SETAF for a subset of ANT's tasks.

4.3. 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 performance analysis, we had to set the log level to a high value (*i.e.*, DEBUG) to produce a 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.

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 captured 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.

```
1  Variables:
2   string server_name;
3
4  Init:
5   server_name = "Tomcat";
6
7  Reset:
8   server_name = "Tomcat";
9
10 DataPoints:
11  string LF1.cid;
12  string LF2.cid;
13  // ...
14  string LF12.cid;
15
16 Relations:
17   LF1.cid->LF2.cid;
18   LF2.cid->LF3.cid;
19   // ...
20   LF11.cid->LF12.cid;
21
22 // Begin adaptation code section
23 On LF1:
24   [cid] = server_name;
25
26 On LF2:
27   [cid] = server_name;
28 // ...
29
30 On LF12:
31   [cid] = server_name;
```

Listing 7: SETAF specification for Apache Tomcat.

Listing 7 therefore highlights a portion of the SETAF specification for Tomcat. As illustrated in this listing, a variable called *cid* is added to all the log formats to expose the hidden relation.

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 of compiler technique (sec)	8.297
Total evaluation time of interpreter technique (sec)	8.329

Table 9: Results for adapting Tomcat system execution trace using UNITE and SETAF.

Table 9 shows the results for comparing the server startup time calculated by UNITE with and without 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 2). 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.

4.4. 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 5 to adapt the system execution trace generated by ActiveMQ.

After executing UNITE with the SETAF adapter for ActiveMQ, we were able to evaluate that average checkpointing time was 115.917 msec for the scenario discussed above. Software 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.

4.5. 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.

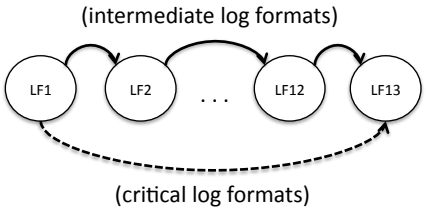

Figure 4: Log formats associated with DAnCE.

As shown in Figure 4, the dataflow model contains 13 different log formats (out of 50+ log formats) that depend on each other. The DAnCE deployment process can be divided into three phases (Deng et al. 2005): (1) deployment preparation phase; (2) start launching phase; and (3) finish launching phase. Each of these phases are driven by remote method calls between different components in DAnCE. The first 5 log formats represent the analysis of the preparation phase. The next 4 log formats represent the start launching phase. The last 4 log formats represent the finish launching phase. If the software testers want to isolate different phases for performance testing, the tester can use log formats within each phase for that purpose.

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 8 therefore presents the SETAF specification for adapting DAnCE's generated system execution traces for analysis using UNITE.

```
1  Variables:
2    string planid_;
3    int lf12_count_, lf13_count_, nodeid_;
4
5  Init:
6    lf12_count_ = lf13_count_ = nodeid_ = 0;
7
8  Reset:
9    lf12_count_ = lf13_count_ = nodeid_ = 0;
10
11 DataPoints:
12   string LF1.planid; string LF2.planid;
13   string LF5.planid; string LF6.planid;
14   string LF9.planid; string LF11.planid;
15   int LF12.nodeid; int LF13.nodeid;
16
```

```
17  Relations:
18    LF1.planid->LF2.planid; LF5.planid->LF6.planid;
19    LF6.planid->LF7.planid; LF8.planid->LF9.planid;
20    LF9.planid->LF10.planid; LF10.planid->LF11.planid;
21    LF11.planid->LF12.planid; LF12.nodeid->LF13.nodeid;
22
23  // Begin adaptation code section
24  On LF1:
25    [planid] = planid_;
26  On LF2:
27    [planid] = planid_;
28  On LF5:
29    [planid] = planid_;
30  On LF6:
31    [planid] = planid_;
32  On LF9:
33    [planid] = planid_;
34  On LF11:
35    [planid] = planid_;
36  On LF12:
37    [nodeid] = lf12_count_;
38    lf12_count_ = lf12_count_ + 1;
39    plan_id_ = [planid];
40  On LF13:
41    [nodeid] = lf13_count_;
42    lf13_count_ = lf13_count_ + 1;
```

Listing 8: SETAF specification for DAnCE.

As illustrated in Listing 8, all the log formats in the SETAF specification for DAnCE, other than `LF12` and `LF13`, use the private variable `planid` to get an adaptation value. The value of this private variable is set by `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 $O(mn)$ runtime complexity where m is the number of log formats in the dataflow model and n is the number of log messages in the system execution trace.

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 10 illustrates the results of analyzing deployment time for each node after adapting DAnCE's generated system execution trace. 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 2) and some newly added log format variables required analyzing other log messages to populate their corresponding value (*i.e.*, Challenge 3 in Section 2). 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.

Node	DT w/o SETAF	DT w/ SETAF (sec)
EC	N/A	5.0
BMDevice	N/A	5.0
BMClosedED	N/A	5.0
BMDisplay	N/A	5.1
Total evaluation time of compiler technique (sec)	N/A	0.237
Total evaluation time of interpreter technique (sec)	N/A	0.272

Table 10: Results for adapting DAnCE's system execution trace using SETAF to measure deployment time (DT).

4.6. Performance Comparison of SETAF Compiled and Interpreted Adaptation techniques

We compared the load time, processing time, total evaluation time and percentage difference in total evaluation time for the compiled and interpreted adapters in SETAF. This performance analysis is based on the following equation:

$$\text{Total Evaluation Time} = \text{Load Time} + \text{Process Time}$$

In the compiled adapter, the load time is the amount of time taken to load the compiled adapter module. In the interpreted adapter, the load time is the amount of time taken to parse the adapter specification and create the object model that represents an adapter. In both cases, the process time is the amount of time taken to evaluate the dataflow model using either adapter.

Figure 5 shows the load times for both the techniques. As shown in the figure, the compiled adapter has a much lower load time when compared to the interpreted adapter. This is expected because the interpreted adapter has to parse the specification and build an object model at run-time. In the compiled adapter, this object model is already in binary form and UNITE only has to load the compiled adapter into memory.

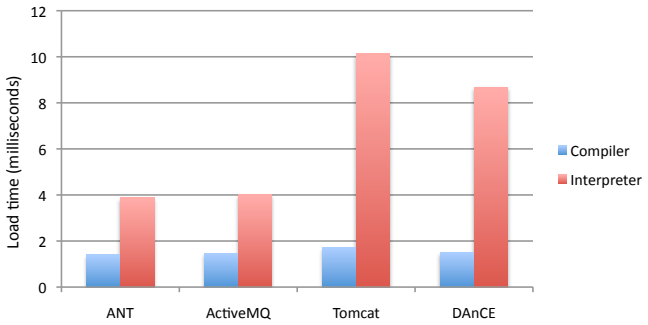

Figure 5: Load times of SETAF interpreted and compiled adapter.

Although the load times are significantly different between the compiled and interpreted adapters, Figure 6 shows that there is not much difference in total evaluation times. This is because total evaluation time for both techniques is dominated by the processing time (*e.g.*, more than 96% as shown in Table 11). The reason for the dominance in processing time is the number of database operations executed during the analysis stage of UNITE.

For example, for each log format defined in the dataflow model UNITE needs to find all the corresponding log message instances (Hill et al. 2009). This needs to be done because UNITE needs to extract values for the data points in each log format . Therefore it needs to iterate through the system execution trace each time it finds a new log format. This complexity is the same for both the compiled and interpreted adapter, and reflected in Figure 6.

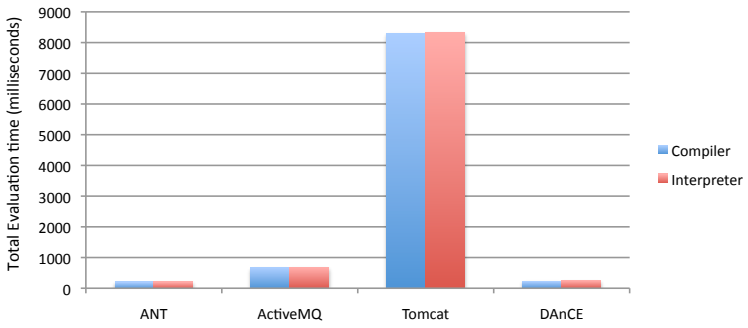

Figure 6: Total evaluation time of SETAF interpreted and compiled adapter.

Open-source Project	**Size of the trace (KB)**	**Compiled Adapter**	**Interpreted Adapter**
ANT	4032	99.33%	98.23%
ActiveMQ	2242	99.79%	99.41%
Tomcat	27880	99.97%	99.87%
DAnCE	576	99.36%	96.80%

Table 11: Process time as a percentage of total evaluation time.

Final discussion. The four case studies previously discussed three different adaptation patterns. The ANT 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 for performance analysis.

The performance comparison results of the two adaptation techniques show that there is little difference between a compiled and interpreted adapter. We

therefore recommend that software system testers use interpreted adapters because as mentioned in Section 3.2 it is easier to use. Moreover, the interpreted adapter provides more flexibility and usability without sacrificing the performance.

4.7. Threats to Validity

One major assumption in these experiments is the collected log messages correctly show the time of occurrence of events. This means system developers must insert the log statement temporally adjacent to the program statement that is responsible for the particular event. If the log statement is temporally faraway from the event statement, then the analysis can give incorrect results. This form of "good practice" is not difficult to adhere when developing software. It is therefore not a big threat for using SETAF and UNITE for performance analysis.

Another important aspect of these experiments is that they are tightly coupled to the generated system execution trace. For example, same system can be executed in different modes, such as with different configuration settings and different operating conditions depending on user requirements.In these situations,the generated system execution trace may differ from mode to mode, and therefore performance analysis results may also differ. Different executions under the same mode, however, will produce similar results, because executions under the same mode always generates the same execution trace but with different timestamps.

5. Related Work

Intrusive instrumentation approaches. Several intrusive approaches can be found in the literature that use traces from instrumented source code for performance analysis (Geimer et al. 2010; Wolf and Mohr 2003; Wylie et al. 2007). Unlike SETAF, these approaches instrument the source code of the target system using methods defined in the performance analysis tool. A main requirement for these approaches therefore is the availability of the source code. Moreover, these approaches add more overhead to the system being analyzed. In contrast, SETAF uses log messages the system outputs during the system execution. This allows SETAF to be used when the source code of the originating system is not available.

Less intrusive instrumentation approaches. TimeToPic (Erkki Salonen 2012) is a tool that can be used to visualize a system execution log. It also provides facilities to analyze different locations, such as points of interests in the visualization graph. This visualization graph can be used to analyze different performance properties. The main limitation in TimeToPic is, it relies heavily on log message format. The developer has to either use the common message format TimeToPic has defined or implement the logger of the application using TimeToPic's logging API. This approach is easy to apply on newly developed systems, and *hard* to apply on existing systems. In contrast, SETAF does not

have any restrictions on the log message format and tries to utilize the system log as much as it can.

(Nagaraj et al. 2012), propose a technique to comparatively analyze system logs to diagnose performance problems. Nagaraj's technique uses two execution logs: one that is called the baseline log and an erroneous log it assumes to have performance related issues. Nagaraj's technique compares the erroneous log with the baseline log and provides a report on the locations of performance problems. The main limitation of Nagaraj's approach is that the perl scripts that process to logs must be modified each time when it encounters a new system. This is because the execution logs of different systems are heterogeneous in nature.

(Han et al. 2012) propose a method called StackMine that uses stack traces collected by the Microsoft® application monitoring tools to debug performance. Their performance monitoring method is less intrusive because the performance monitoring tools send traces only when the response time of a method is above a certain threshold. Furthermore, Han's analysis is offline. The main difference between StackMine and SETAF is that StackMine is operating system dependent because the performance monitoring tools they use are platform dependent. There are other less intrusive approaches for performance monitoring that are language and architecture dependent. For example, previous research efforts have focused on proposing methods for performance monitoring in Enterprise Java Beans (EJB) applications (Mos et al. 2001; Mania et al. 2002; Parsons et al. 2006). These approaches, however, require users to deploy performance monitoring beans.

Other analytical domains. (Cinque et al. 2009), propose a technique for dependability evaluation of complex software systems using system execution traces. In their approach, they propose to follow a minimal set of logging rules during the design and development time. This set of rules guarantees that the system logs contain the required properties for dependability analysis of complex systems. For example, the main types of rules that Cinque proposes are enabling logging for service start/end and interaction start/end events. Even though this approach contributes to the generation of well structured logs, it is highly unlikely that every system follows these rules. Current software development methodologies highly depend on software reuse. It therefore is highly unlikely to assume that this kind of structure is followed in complex software systems. Instead, SETAF tries to tackle this problem of uncertainty in system execution traces using the adaptation technique.

(Yang et al. 2006), use system execution traces for dynamic inference. Similar to SETAF, Yang also accepts the fact that system execution traces are imperfect. Likewise, Yang uses coding conventions to prune large number of unimportant properties for developers. SETAF, however, does not rely on such coding conventions. Instead, SETAF provides a framework to capture execution semantics that are not reflected in the execution trace.

Table 12 summarizes the above mentioned approaches for performance analysis using system execution traces. In this table we are summarizing the approaches described in this section based on the following four characteristics.

- **Intrusive** - Modifications requires to the system for the purpose of performance analysis. The possible values are *High* and *Low*.
- **Requires Source** - The approach requires the source code for the analysis. The possible values are *Yes* and *No*.
- **Architecture Dependent** - The approach works only for systems that use a specific architecture (*e.g.* J2EE, .NET). The possible values are *Yes* and *No*.
- **Restricted Log Format** - The approach assumes a particular structure in the log statements. The possible values are *Yes* and *No*.

If a characteristic is not applicable to a particular approach, then it is denoted as "Not Applicable" (N/A).

	Instrusive	**Requires Source**	**Arch. Dependent**	**Restricted Log Format**
Geimer 2010	High	Yes	No	N/A
Wolf and Mohr 2003	High	Yes	No	N/A
Wylie 2007	High	Yes	No	N/A
TimeToPic	Low	No	No	Yes
Nagaraj 2012	Low	No	No	No
StackMine	Low	No	Yes	N/A
Murphy 2006	Low	No	Yes	No
Mos 2001	Low	No	Yes	No
Mania 2002	Low	No	Yes	No
SETAF	Low	No	No	No

Table 12: Summary Table on related work.

6. Concluding Remarks

System execution traces are information rich resources that can be used to understand system properties, such as its performance properties. This, however, is only possible if the system execution trace contains properties required 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 performance analysis. SETAF operates by allowing system testers to write adaptation pattern specifications for an existing system execution trace. These adaptation specifications can then be used with UNITE to analyze performance properties correctly. As shown in the results

from applying SETAF to several open-source projects, it is possible to perform such adaptation without modifying the original source code.

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

- **Automatically identifying the dataflow model from a system execution trace.** Complex software systems, such as distributed systems can easily generate system execution traces that are quite large. 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 performance analysis. Future work therefore includes applying existing data mining techniques to assist in identifying the dataflow model—thereby easing complexities associated with the analysis. Furthermore, with adequate domain knowledge, data mining techniques can also be used to assist in locating adaptation patterns, which are currently done manually by the system tester.

- **Adaptation specification size does not have much impact on evaluation time of performance analysis.** As learned from the performance comparison of the compiled and interpreted adapters, total evaluation time depends on the number of log messages and the size of the dataflow model (*i.e.*, processing time). For example, total evaluation time for Apache Tomcat is greater than DAnCE even though DAnCE's adaptation specification is larger. This is because Apache Tomcat's system execution trace has more log messages when compared to DAnCE's system execution trace.

- **Correctness of the adaptation specification.** As stated above, system testers have to manually analyze the dataflow models and write the SETAF adaptation specifications. The performance analysis results from UNITE and SETAF therefore cannot be guaranteed if testers do not analyze the model correctly and write a specification correctly. Although this is true, the main focus of SETAF is to provide the framework for writing adaptation specifications, and support UNITE when analyzing system execution trace.

  SETAF therefore guarantees syntactical correctness, but not adaptation correctness. This is similar to a compiler that guarantees syntactical correctness, but does not guarantee execution correctness because that depends on the developer implementing correct logic. Similarly, SETAF provides a domain-specific language to specify adaptation specifications for system execution traces and the correctness of the adaptation highly depends on (1) the domain-knowledge of the system tester and (2) how well the system tester writes the adaptation specification.

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

References

Allen, F. E., Cocke, J., March 1976. A program data flow analysis procedure. Commun. ACM 19, 137–.

Bell, T., 1999. The Concept of Dynamic Analysis. In: Proceedings of the 7th European Software Engineering Conference. Springer-Verlag, London, UK, pp. 216–234.

Breu, S., Krinke, J., 2004. Aspect mining using event traces. In: Automated Software Engineering, 2004. Proceedings. 19th International Conference on. IEEE, pp. 310–315.

Chappell, D., 2004. Enterprise Service Bus. OReilly.

Cinque, M., Cotroneo, D., Pecchia, A., 2009. A logging approach for effective dependability evaluation of complex systems. In: Proceedings of the 2009 Second International Conference on Dependability. DEPEND '09. IEEE Computer Society, Washington, DC, USA, pp. 105–110.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R., 2009. A systematic survey of program comprehension through dynamic analysis. Software Engineering, IEEE Transactions on 35 (5), 684–702.

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

Erkki Salonen, R. P., 2012. Find the bug, Fix the bug, Do it fewer times (TimeToPic). http://www.timetopic.net/Pages/default.aspx.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1997. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley.

Geimer, M., Wolf, F., Wylie, B., Ábrahám, E., Becker, D., Mohr, B., 2010. The scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22 (6), 702–719.

Han, S., Dang, Y., Ge, S., Zhang, D., Xie, T., 2012. Performance debugging in the large via mining millions of stack traces. In: Proceedings of the 2012 International Conference on Software Engineering. IEEE Press, pp. 145–155.

Hill, J. H., August 2010. Context-based Analysis of System Execution Traces for Validating Distributed Real-time and Embedded System Quality-of-Service Properties. In: Proceedings 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). Macau, P.R.C., pp. 92–101.

Hill, J. H., Turner, H. A., Edmondson, J. R., Schmidt, D. C., apr 2009. 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, pp. 406–415.

Mania, D., Murphy, J., McManis, J., 2002. Developing performance models from non-intrusive monitoring traces. Proceeding of Information Technology and Telecommunications (IT&T).

Mos, A., Murphy, J., et al., 2001. Performance monitoring of java component-oriented distributed applications. In: Proc. IEEE 9th International Conference on Software, Telecommunications and Computer Networks (SoftCOM). pp. 9–12.

Nagappan, M., Wu, K., Vouk, M. A., 2009. Efficiently extracting operational profiles from execution logs using suffix arrays. In: Proceedings of the 20th IEEE International Conference on Software Reliability Engineering. ISSRE'09. pp. 41–50.

Nagaraj, K., Killian, C., Neville, J., 2012. Structured comparative analysis of systems logs to diagnose performance problems. In: Symposium on Networked Systems Design and Implementation. USENIX Association. pp. 26–26.

Object Management Group, Jul. 2003. Deployment and Configuration Adopted Submission. Object Management Group, OMG Document mars/03-05-08 Edition.

Parsons, T., Mos, A., Murphy, J., 2006. Non-intrusive end-to-end runtime path tracing for j2ee systems. IEE Proceedings Software 153 (4), 149.

Peiris, T. M., Hill, J. H., April 2012. Adapting System Execution Traces for Validation of Distributed System QoS Properties. In: Proceedings of 15th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC). Shenzhen, China, pp. 162–171.

Safyallah, H., Sartipi, K., 2006. Dynamic analysis of software systems using execution pattern mining. In: Proceedings of the 14th IEEE International Conference on Program Comprehension. pp. 84–88.

Schmidt, D., 1993. The ADAPTIVE Communication Environment: An object-oriented network programming toolkit for developing communication software.

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

Silverstein, C., Marais, H., Henzinger, M., Moricz, M., September 1999. Analysis of a Very Large Web Search Engine Query Log. SIGIR Forum 33, 6–12.

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

SUN, 2002. Java Messaging Service Specification. `java.sun.com/products/jms/`.

Sun Micro Systems, May 2006. Java Server Pages Specification . SUN, Version 2.1 Edition.

Sun Micro Systems, Dec. 2009. Java Servlet Specification . SUN, Version 3.0 Edition.

Wolf, F., Mohr, B., 2003. Automatic performance analysis of hybrid mpi/openmp applications. Journal of Systems Architecture 49 (10), 421–439.

Wylie, B., Wolf, F., Mohr, B., Geimer, M., 2007. Integrated runtime measurement summarization and selective event tracing for scalable parallel execution performance diagnosis. Applied Parallel Computing. State of the Art in Scientific Computing, 460–469.

Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M., 2008. Mining Console Logs for Large-scale System Problem Detection. In: Proceedings of the Third conference on Tackling Computer Systems Problems with Machine Learning Techniques. SysML'08. Berkeley, CA, USA, pp. 4–4.

Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M., 2006. Perracotta: mining temporal api rules from imperfect traces. In: Proceedings of the 28th international conference on Software engineering. ICSE '06. ACM, New York, NY, USA, pp. 282–291.

Yin, Y., Byna, S., Song, H., Sun, X., Thakur, R., 2012. Boosting application-specific parallel i/o optimization using iosig. In: Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on. IEEE, pp. 196–203.