Auto-Constructing Dataflow Models from System Execution Traces

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

***Abstract*—This paper presents a method and tool named the *Dataflow Model Auto-Constructor (DMAC)*. DMAC uses frequent-sequence mining and Dempster-Shafer theory to mine a system execution trace and reconstruct its corresponding dataflow model. Distributed system testers then use the resultant dataflow model to analyze performance properties (*e.g.*, end-to-end response time, throughput, and service time) captured in the system execution trace. Results from applying DMAC to different case studies show that DMAC can reconstruct dataflow models that cover at most 94% of the events in the original system execution trace. Likewise, more than 2 sources of evidence are needed to reconstruct dataflow models for systems with multiple execution contexts.**

***Index Terms*—DMAC, system execution traces, dataflow models, auto-construction, frequent-sequence mining, quality-of-service, evidence theory, domain knowledge**

I. Introduction

System execution traces [1], which are a collection of log messages, are useful artifacts for analyzing distributed systems. They have been used in system failure detection [2], operational profiling [3], and usage analysis of websites [4]. System execution traces have also been used to validate performance properties (*e.g.*, end-to-end response time, throughput, and service time) [5].

For example, *Understanding Non-functional Intensions via Testing and Experimentation (UNITE)* [5] is a tool for analyzing user-defined performance properties via system execution traces. UNITE's analysis of a system execution trace is driven by a dataflow model—a directed acyclic graph of cause-effect relationships among different event types in the system (not event instances). Using the dataflow model, UNITE is able to reconstruct execution flows in the system and support correct analysis of user-defined performance properties.

Although UNITE addresses challenges associated with analyzing distributed system performance properties via system execution traces, it is both tedious and time-consuming for distributed system testers to define the required dataflow model. Likewise, as distributed systems increase in both size (*i.e.*, source lines of code and number of software/hardware components) and complexity (*i.e.*, set of features), it becomes *harder* to manually define the dataflow model. This is because distributed system testers have to analyze the entire system execution trace and its originating source code, which is not always readily available or easy to understand.

Because a dataflow model is associated with a system execution trace and a system execution trace is a sequential occurrence of events that should be consistent across different executions, it should be possible to automatically reconstruct a dataflow model from its corresponding system execution trace. This would alleviate challenges associate with defining the dataflow model. Moreover, automatically reconstructing dataflow models will reduce disconnect between a dataflow model and its corresponding system execution trace being analyzed—especially as system implementation evolves.

This paper therefore presents a method and a tool named *Dataflow Model Auto Constructor (DMAC)*. DMAC automatically reconstructs dataflow models from system execution traces by (1) using an iterative technique based on frequent-sequence mining [6] to correctly distinguish between variable and static parts of log messages (*i.e.*, identify log formats) in a system execution trace; and (2) using Dempster-Shafer theory [7], which is a mathematical theory of evidence, to identify causality between different log formats. The resultant dataflow model can then be used within UNITE to evaluate performance properties of a software system (*e.g.*, execution time, service time, and response time).

Experimental results show that DMAC can reconstruct dataflow models that cover at most 94% of the events in the original system execution trace. Likewise, more than 2 evidences are needed to reconstruct dataflow models when the originating system that generated the system execution trace has parallel executions.

Paper organization. The remainder of this paper is organized as follows: Section II provides an overview of UNITE; Section III discusses the design and functionality of DMAC; Section IV presents results from applying DMAC to several open-source projects; Section V discusses work related to DMAC; and Section VI provides concluding remarks and lessons learned.

II. Overview of UNITE and its Limitations

Because UNITE is a new tool, we are providing brief overview of its core concepts. Table I illustrates an example system execution trace. Using the information in Table I,

This work 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.

ID	Time of Day	Hostname	Severity	Thread	Message
1	2011-04-25 13:15:56	senderA.cs.iupui.edu	INFO	1	A sent message 1 at 10
2	2011-04-25 13:16:10	senderA.cs.iupui.edu	INFO	1	A received message 1 at 20
3	2011-04-25 13:16:20	senderA.cs.iupui.edu	INFO	1	Got the authentication for request at 30
4	2011-04-25 13:16:30	senderB.cs.iupui.edu	INFO	2	B sent message 2 at 40
5	2011-04-25 13:16:40	senderB.cs.iupui.edu	INFO	2	B received message 2 at 50
6	2011-04-25 13:16:55	senderB.cs.iupui.edu	INFO	2	Access denied at 60
7	2011-04-25 13:17:10	senderC.cs.iupui.edu	INFO	3	C sent message 3 at 70
8	2011-04-25 13:17:21	senderC.cs.iupui.edu	INFO	3	C received message 3 at 80
9	2011-04-25 13:18:35	senderC.cs.iupui.edu	INFO	3	Got the authentication for request at 90

distributed system testers first create a dataflow model of the system execution trace as shown in Listing 1.

```
LF1: {STRING cid} sent message {INT msgid} at {INT sent}
LF2: {STRING cid} received message {INT msgid} at {INT recv}
LF3: Got the authentication for request at {INT auth}
LF4: Access denied at {INT denied}

Relations:
  LF1->LF2, LF2->LF3, LF2->LF4
  LF1.cid = LF2.cid; LF1.msgid = LF2.msgid
```

Listing 1. Dataflow model for system execution trace in Table I.

As shown above, the dataflow model contains four log formats (*i.e.*, `LF1`, `LF2`, `LF3` and `LF4`) and a relation between them based on their variable parts (*i.e.*, the words between brackets in the log formats). Testers then define an expression for analyzing a corresponding performance property based on variable parts in the dataflow model. For example, `AVG(LF2.recv - LF1.sent)` evaluates average latency of events based on data captured in the system execution trace above. If the aggregation function is removed, then UNITE produces a data trend.

For trivial system execution traces, it is not hard to manually define a dataflow model. This, however, is not the case for large and complex system execution traces. In this situation, it is ideal to automatically reconstruct a dataflow model from its corresponding system execution trace. Based on this need, we have identified the following challenges for automatically reconstructing a dataflow model:

- **Challenge 1: Correctly identifying valid log formats.** Log formats contain both static and variable parts. The variable parts are used to define causal relations between log formats and define expressions that evaluate a performance property. Failure to identify the correct static and variable parts can reduce the number of possible variables available for usage in an expression. Moreover, it can inhibit our ability to define correct and complete relations in the dataflow model. It is therefore important to correctly identify complete log formats to create a comprehensive dataflow model.
- **Challenge 2: Correctly identifying causal relations between log formats.** When all the events are in a single execution context (*i.e.*, no concurrency), we can assume causality is defined by the order of occurrences for the events. This is because the representative log messages generated for these events occur from a source code that is executed serially. In this case we can generate the dataflow model based on their execution order. The difficult task, however, is identifying cause-effect relationships between events that occur in different execution contexts because it introduces multiple local clocks, which may not be synchronized. Even if we assume a single clock (*i.e.*, a global clock), there may, or may not, be explicit relationships between events that occur in different execution contexts, such as an identifier for associating related events.
- **Challenge 3: Correctly identifying relationships between variables.** After identifying the log formats and relations of the dataflow model, one must identify relations between the variables such that their values are equal for all the instances. Similar to the previous challenges, it is important to identify correct relations between variables to ensure correct correlation of data points.

The remainder of this paper discusses how DMAC addresses the above challenges when auto-constructing dataflow models.

III. THE DESIGN AND FUNCTIONALITY OF DMAC

Figure 1 illustrates DMAC's workflow for reconstructing a dataflow model from a system execution trace. As shown in this figure, the process consists of two major steps: (1) identifying log formats for the dataflow model of the corresponding system execution trace; and (2) identifying causal relationships between different log formats.

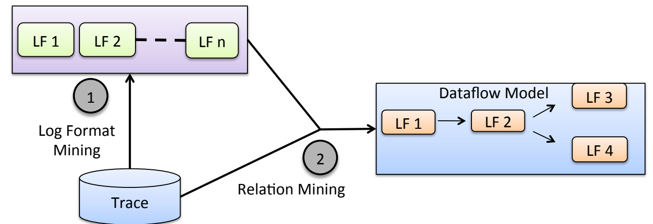

Fig. 1. DMAC's workflow for reconstructing a dataflow model.

The log format mining step (*i.e.* step 1) is based on frequent-sequence mining. This section therefore gives a brief overview of frequent-sequence mining, then describes the functionality of DMAC's log format mining algorithm.

A. Overview of Frequent-Sequence Mining

Given a collection of sequences S and a support threshold $\sigma \in (0, 1]$, frequent-sequence mining locates all the sequences

that are sub-sequences of at least $\sigma \times |S|$ sequences. The support of a sequence t is defined as the number of sequences in S such that t is a sub-sequence. Mathematically,

$$\sum_{\delta \in S} I(t \sqsubseteq \delta) \quad (1)$$

where $\sqsubseteq$ denotes a sub-sequence and I is an indicator random variable. Lastly, the support threshold σ is called *minimum support* and denoted by *min-sup*.

In the context of system execution traces, we define sequences as words separated by a delimiter (*e.g.*, space, tab, and newline). There are several methods for frequent-sequence mining, *e.g.*, *Generalized Sequential Pattern (GSP)* [8] and *Sequential PAttern Discovery using Equivalence classes (SPADE)* [9]. DMAC uses *SPADE* because of its efficiency [9].

DMAC considers all *k-word* sequences where $k > 1$ because DMAC performs multiple levels of mining (discussed later) to select candidate sequences and remove sub-sequences from larger ones. Moreover, most log formats have at least two words as their static part. Considering two or more word sequences therefore is sufficient because most single word sequences appear in two or more word sequences as sub-sequences. Likewise, single word sequences that are not part of any higher-order sequence should correspond to one word log formats.

To better understand how frequent-sequence mining relates to DMAC, applying frequent-sequence mining with *min-sup* value of 0.33 to the log messages in Table I in Section II produces the set of sequences shown in Listing 2. Reducing the *min-sup* to 0.2 will generate all the combinations with `A,B,C,1,2,3` in addition to the set of frequent-sequences shown in Listing 2.

```
sent message, sent at, message at, received message,
received at, sent message at, received message at
```

Listing 2. Frequent-sequences for *min-sup* value of 0.33.

DMAC then uses the resulting frequent-sequences to identify log formats of a system execution trace. When an event occurs multiple times during the system execution, its corresponding log message will appear in the system execution trace with variations in its variable parts; whereas, its static parts do not change. This implies that the static parts have a relatively high frequency compared to its variable parts. DMAC therefore assumes frequent-sequences correspond to words in static parts of a log format.

Because of DMAC's assumption it is important to select a relevant *min-sup* value. Finding the appropriate *min-sup* value is a challenging problem in the field of data mining [10]. If the min-sup is set to a very low value, variable parts of a log format will be filtered as static parts (as discussed in the example above). Likewise, if the *min-sup* is set to high, then SPADE will generate a smaller set of frequent-sequences. This can cause DMAC to miss log formats. DMAC therefore uses an iterative process that enables it to start with a relatively high *min-sup* value and still identify a large number of log formats present in the system execution trace.

B. Identifying log formats.

Identifying the set of log formats in a dataflow model is a challenging process. This is because it requires differentiating static parts from variable parts, and identifying their correct positions within the log format. For a given system execution trace, finding the set of log formats is a two step process. In the first step, DMAC uses a user-provided *min-sup* value. This step is similar to the process described in Section III-A.

The next step is constructing the log formats from the identified frequent-sequences. Based on the assumption described in Section III-A where frequent-sequences are candidate static parts of the log formats, DMAC uses the frequent-sequences to identify the variable parts of a log format. More specifically, DMAC uses the following sets to assist with constructing valid log formats from identified frequent-sequences:

- **Frequent-sequences** (F). The set F is the frequent-sequences generated by SPADE for a given *min-sup* value. For example, Listing 2 is the set F for Table I with *min-sup* value of 0.33.
- **Maximal-sequences** (M). The set M is the maximal-sequences for F. A set of frequent-sequences is called *maximal* if it is not a subset of any other set of frequent-sequences. For example, Listing 3 shows the set M for Listing 2. Maximal sequences are important because a maximal sequence contains all the static parts for its corresponding log format.

```
sent message at
received message at
```

Listing 3. Maximal-sequence set for Listing 2.

- **Position vector** (p). The position vector p is an integer vector that tracks the position numbers of words in a maximal sequence (*i.e.*, i^{th} word of the log message) that appear in an actual log message. A position vector is always associated with a maximal sequence $m \in M$. Each value in p represents the position of the corresponding word of its associated maximal sequence m when m appears in a log message. There can also be different position numbers for m in different log messages. Listing 4 shows the position vector for M in Listing 3. We denote the set of all position vectors associated with a maximal sequence m as p_m.

```
sent message at - (2, 3, 5)
received message at - (2, 3, 5)
```

Listing 4. Position vectors for the maximal sequences in Listing 3

DMAC first uses SPADE to construct the set F. DMAC then executes the following steps to construct log formats for a system execution trace:

1) For the set F, DMAC first generates the set M.
2) $\forall m \in M$, DMAC calculates the set of position vectors p_m. DMAC does this by checking if m is a sub-sequence of any log message in the system execution trace. If it is a sub-sequence of a log message and $p \notin p_m$, then a new position vector p is created and added to the set p_m. When a new p is added to p_m, the size of the log

message is also recorded. This is used to construct the log formats as explained next.

3) $\forall m \in M$, log formats are created for each member of p_m. As shown in Figure 2, a dummy log format is first created for the message where each word in the dummy log format is initialized with an empty placeholder ($\{\}$). The number of words in the dummy log format is equal to the number of words recorded in previous step. Likewise, the position vector contains the positions of the static parts and the actual words for the static parts are contained in m. Placeholders that have not been replaced with actual words at the end of this process are assumed to be the log format's variable parts.

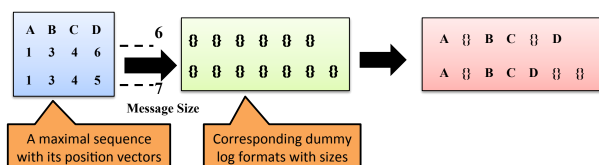

Fig. 2. Log format construction from a maximal-sequence.

Algorithm 1 Algorithm for constructing log formats from system execution traces.

1: **procedure** FINDLOGFORMATS(D, R, σ)
2: D: The system execution trace (Initial system execution trace)
3: R: Required coverage
4: σ: Initial min-sup
5: L: Final log formats
6: $|D|$: Number of messages in D
7:
8: $d \leftarrow D, L \leftarrow \emptyset, C \leftarrow 0$
9: $i \leftarrow 1, \mu \leftarrow \sigma$
10:
11: **while** $C \leq R$ or $i \leq MaxIterations$ **do**
12: $\mu \leftarrow \sigma \times \frac{1}{i} \times \frac{|D|}{|d|}$
13: $F \leftarrow MineSequences(d, \mu)$
14: $l \leftarrow FindLogFormats(d, F)$
15: $d \leftarrow CreateNextSystemExecutionTrace(d, l)$
16: $L \leftarrow L \cup l$
17: $C \leftarrow C + CalculateCoverage(d, l, D)$
18: $i \leftarrow i + 1$
19: **end while**
20: **return** L
21: **end procedure**

Algorithm 1 highlights the steps discussed above. During the first iteration, DMAC uses SPADE to mine the system execution trace. The identified frequent-sequences are used to construct candidate log formats. The candidate log formats are used to prune the current system execution trace—generating a new one (line 15). The new system execution trace is then used in the next iteration. This process continues until a satisfiable coverage percentage (*i.e.*, a measure of the number of log messages covered by the identified log formats) is met or a predetermined number of iterations is exceeded.

One challenge in the iterative process is calculating the *min-sup* value for the upcoming iteration. DMAC addresses this challenge by dividing the initial *min-sup* value by the current iteration number and scaling it to a higher value by multiplying it from the ratio of the sizes of the original system execution trace to the pruned one (line 12). This guarantees that mining process starts from a relatively higher *min-sup* value in each iteration.

DMAC's iterative process has two advantages over using a single iteration with a lower *min-sup* value. First, it reduces the possibility of variable parts being selected as static parts in the final set of log formats. This is because starting with a higher *min-sup* value and pruning log messages after each iteration guarantees that lower *min-sup* values are used on a smaller system execution traces. On the other hand, if DMAC uses a single, low *min-sup* value, then DMAC runs the risk of selecting variable parts as static parts. This issue is reflected in Nagappan et al. [11] approach when we integrated it into DMAC for identifying log formats.

Secondly, it enables achieving a high coverage. This is because the iterative mining process increases coverage after each iteration since infrequent sequences in an iteration become frequent in successive iterations.

C. Mining causal relationships.

The end goal of mining causal relationships is to find the cause-effect graph from a given set of log formats. Given a pair of log formats ($LF_i \rightarrow LF_j$), they can either occur in the same execution context or in two different execution contexts. This facts causes DMAC to use a two step process when mining causal relationships.

Step 1. In the first step, DMAC identifies causal relationships among log formats that occur in the same execution context using the algorithm shown in Algorithm 2.

The steps in Algorithm 2 can be summarized as follows:

1) For each execution context, all its log messages are compared with the identified log formats to produce an execution order, such as `<LF1, LF2, LF3, LF1, ..., LF3>`. DMAC only analyzes adjacent log format pairs in the execution order because it is possible to order events using transitivity. For example, if ($LF1 \rightarrow LF2$) and ($LF2 \rightarrow LF3$) are valid relations, then ($LF1 \rightarrow LF3$) is a valid relation.
2) For each adjacent log format pair in the execution order (*i.e.*, OL_i and OL_{i+1}) the earliest position of OL_i in the execution order should always be less than the earliest position of OL_{i+1} to be considered a valid relation. This ensures no cycles exists and directed acyclic graph properties are not violated.
3) Finally, the `ExtendGraph` function in line 14 adds the log format pair to the graph if they are not already in the graph and do not have an edge between them.

To provide a better understanding of the steps above, let us reconsider the system execution trace from Table I from

Algorithm 2 Algorithm for mining causal relationships in a single execution context.

```
1: procedure MINECAUSALONE(D, G, LF, E)
2:     D: The system execution trace (Initial system execution trace)
3:     G: Directed Acyclic Graph
4:     LF: Final log formats
5:     E: Current Execution Entity
6:     OL: Execution Order
7:
8:     OL ← CreateLogFormatOrderList(E, LF, D)
9:
10:    for all (OL_i, OL_{i+1}) ∈ OL do
11:        m ← FirstPositionOf(OL_i)
12:        n ← FirstPositionOf(OL_{i+1})
13:        if m ≤ n then
14:            ExtendGraph(OL_i, OL_{i+1}, G)
15:        end if
16:    end for
17: end procedure
```

Section II. For this example we assume that they are generated from the same execution context. This system execution trace produces an execution order similar to the following: $< LF1, LF2, LF3, LF1, LF2, LF4, LF1, LF2, LF3 >$. In this case, DMAC will identify ($LF_1 \rightarrow LF_2$), ($LF_2 \rightarrow LF_3$), ($LF_2 \rightarrow LF_4$) as valid relations. DMAC, however, will not identify ($LF_3 \rightarrow LF_1$) and ($LF_4 \rightarrow LF_1$) as valid relations because it forms a cycle.

Step 2. In the second step, DMAC identifies causal relationships among log formats that occur in different execution context. This, however, is a challenge when compared to Step 1 because any two events that happen in different execution contexts can have a causal relationship between each other [12]. This implies that there is a level of uncertainty associated with causal relationships between events that occur in different execution contexts.

To address this challenge, DMAC uses a probabilistic approach since probabilistic frameworks are most suited for causality mining [13]. More specifically, DMAC uses Dempster-Shafer (DS) theory [7], which is is a mathematical theory of evidence, to mine causal relationships for log formats that occur in different execution contexts.

In DS theory, a set of mutually exclusive and exhaustive set of hypothesis are referred to as Frame of Discernment (*FoD*). For example, $\{(LF_i \rightarrow LF_j) = Yes, (LF_i \rightarrow LF_j) = No\}$ is the *FoD* in mining causal relations. The *basic belief assignment, (bba)* function, also called the *mass distribution function* m, distributes belief over the power set of the *FoD*. Likewise, the belief function (Bel) is a measure of how much confidence we have for a certain hypothesis to be true; whereas the *bba* specifies the weight (mass) of a particular evidence source has to support a given hypothesis.

For example, $m((LF_i \rightarrow LF_j) = Yes)$ defines evidence for supporting the relationship for a particular source of evidence, and $m((LF_i \rightarrow LF_j) = No)$ defines evidence for disqualifying the causal relation for a particular evidence source. $m((LF_i \rightarrow LF_j) = Yes, (LF_i \rightarrow LF_j) = No)$ is the measure of uncertainty, or the ignorance, that a particular evidence source has about the relation. Unlike traditional probability theory if $m((LF_i \rightarrow LF_j) = Yes) = p$, then it does not necessarily mean that $m((LF_i \rightarrow LF_j) = No) = 1 - p$. This is because sources of evidence only support $\{(LF_i \rightarrow LF_j) = Yes\}$ and its ignorance about $\{(LF_i \rightarrow LF_j) = No\}$ should be assigned to $\{(LF_i \rightarrow LF_j) = Yes, (LF_i \rightarrow LF_j) = No\}$.

Since there are many sources of evidences for a particular proposition, DS theory provides Dempster's rule to combine evidences and calculate a single belief. For example, after combining all the evidences, $Bel((LF_i \rightarrow LF_j) = Yes)$ specifies the confidence we have to support the causal relation between LF_i and LF_j. In general, Dempster's rule can be used in situations where there are conflicting evidences from different sources. DMAC therefore uses the Dempster's rule to combine evidences and handle conflicting evidences. Algorithm 3 illustrates the major steps DMAC uses to apply DS theory.

Algorithm 3 Algorithm for mining causal relationships between log formats in different execution entities.

```
1: procedure MINECAUSALTWO(G, LF, EL, λ)
2:     G: Directed Acyclic Graph
3:     LF: Final log formats
4:     E: Execution Entity List
5:     λ: Belief Threshold
6:     B: Belief value
7:
8:     for all LF_i, LF_j do
9:         if HasCommonEE(LF_i, LF_j, E) then
10:            Continue
11:        end if
12:        if AreReachable(LF_i, LF_j, G) then
13:            Continue
14:        end if
15:        B ← CalculateBelief(LF_i, LF_j)
16:        if B ≥ λ then
17:            ExtendGraph(LF_i, LF_j, G)
18:        end if
19:    end for
20: end procedure
```

As shown above, DMAC first verifies if the two log formats occur in different execution contexts (line 9). DMAC then checks if the corresponding nodes for these log formats in the current graph are reachable from each other (line 12). This is necessary because it implies a transitive relationship.

Lastly, DMAC evaluates evidence from different sources and combines it using Dempster's rule to associate a single belief value $b \in [0, 1]$ to any log format pair (line 15). The belief value is a measure of confidence about the causal

relation between the two log formats. Currently, DMAC uses the following three evidence sources to calculate the belief value of a particular causal relation $LF_i \rightarrow LF_j$:

1) **Time evidence.** Time evidence is based on the observation that two events can be causally related if they occur within a certain time window [14]. To calculate the time evidence value, DMAC first checks whether there is at least one LF_j log message occurs within the time window for each LF_i log message, and calculates a ratio of identified LF_j to the number of LF_i log messages. The ratio is then multiplied by a weight based on the ratio between actual time difference between the two events and the time window, that reflects how "easy" for two events satisfy the time window requirement. For example, the closer the time difference is to 0, the closer the weight is to 1.
Time evidence only assigns a *bba* to the $\{(LF_i \rightarrow LF_j) = Yes\}$ hypothesis. Failing to satisfy the timing window does not necessarily mean $\{(LF_i \rightarrow LF_j) = No\}$. DMAC therefore assigns $1-p$ value to the hypothesis $\{(LF_i \rightarrow LF_j) = Yes, (LF_i \rightarrow LF_j) = No\}$ Lastly, when calculating the time evidence for any candidate causal relation $\{(LF_i \rightarrow LF_j) = Yes\}$, we assume that clocks in the different execution contexts are synchronized. If there is high clock drift between the different execution contexts, then we decrease the calculated evidence value by multiplying it with a weight that decreases the confidence about this event source.
2) **Variable evidence.** Variable evidence is based on the observation that $LF_i \rightarrow LF_j$ can be true if they both have variable parts that match across many occurrences of the log messages. Based on this observation DMAC calculates the *bba* for the hypothesis $\{(LF_i \rightarrow LF_j) = Yes\}$ using the approach presented in Algorithm 4.
As shown in this algorithm, if LF_i has m variables and LF_j has n variables, then for any two messages it requires $m \times n$ comparisons. The comparison of variables can be computationally expensive, but most log formats do not have more than 4 variable. Within the $m \times n$ iterations, the algorithm checks whether the values of the two variables are equal (line 13). If the variable values are equal, then the algorithm increments the counter that tracks the number of of instances satisfying the variable relation candidate in line 13. After iterating over all the variable parts, the algorithm outputs portions of the messages that satisfy the log format variable relationship. Lastly, the algorithm calculates the *bba* for $\{(LF_i \rightarrow LF_j) = Yes\}$ hypothesis using the maximum count of the $m \times n$ variable relation candidates (line 19).
3) **Domain-specific evidence.** The two evidence sources previously discussed are common across all system execution traces. There can be cases when time and variable evidence is not enough to construct a valid dataflow model. When this situation occurs, we rely on domain-knowledge to integrate domain-specific evidences (*i.e.*, the evidence that pertains only to the dataflow model and corresponding system execution trace). Because of the need to incorporate domain-knowledge into DMAC, we have created a framework that allows testers to integrate their own domain-knowledge about the system execution trace as another evidence source.
Figure 3 shows the general architecture for how domain-specific evidence is integrated into DMAC. As shown in this figure, the user can specify domain-specific evidence at the DMAC-level, or the user-level, which is converted to DMAC-level evidences. At the user-level, the tester specifies knowledge about causal relationships between log messages using a natural language and associates a quantitative value between 0 and 1 with each piece of knowledge. For example, ("*send*" $\rightarrow$ "*receive*", 0.6)

Algorithm 4 Algorithm for calculating Variable evidence.

```
1:  procedure CALCULATEVAREVIDENCE(LF_i, LF_j)
2:      LF_i: Candidate Cause log format
3:      LF_j: Candidate Effect log format
4:      V_i: Set of variable values of LF_i
5:      V_j: Set of variable values of LF_j
6:      M: Map of {(v_m, v_n), c}
7:      v_m ∈ V_i, v_n ∈ V_j
8:      c: Count of each matching (v_m, v_n)
9:
10:     for all LF_i messages do
11:         for all LF_j messages do
12:             for all (v_m, v_n) do
13:                 if v_m = v_n then
14:                     increment corresponding c in M
15:                 end if
16:             end for
17:         end for
18:     end for
19:     S ← Max(c)
20:     n ← |LF_i|
21:     return S/n
22: end procedure
```

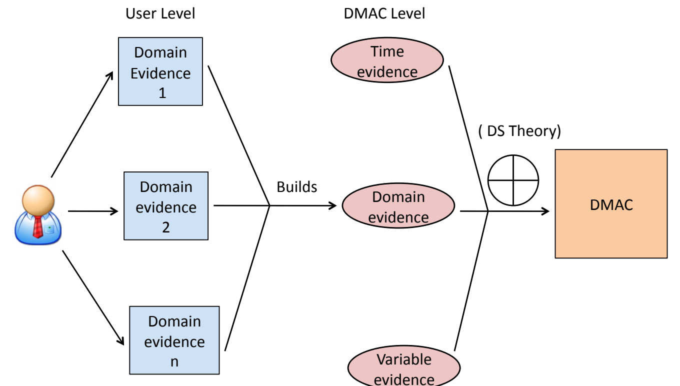

Fig. 3. Combing evidences for causal relation mining process.

means that corresponding log formats that match log messages with either "send" or "receive" are causally related with 0.6 certainty. At the DMAC-level, the tester specifies knowledge about causal relationships between log formats. For example, $(LF_i \rightarrow LF_j, 0.6)$ means that LF_i causes LF_j with 0.6 certainty.

DMAC then uses the DMAC-level domain-specific evidence—along with the time and variable evidence—to auto-construct the dataflow model. It is worth noting that domain-specific evidence is not required for DMAC to work correctly. It, however, is useful to incorporate domain-specific evidence if the time and variable evidences are not producing an accurate dataflow model. Section IV discusses results related to this observation.

D. Identifying causal relationships among variables.

The final part of constructing the dataflow model is identifying relationships between variable parts of different log formats. For each log format variable, DMAC keeps all the values extracted from its corresponding log messages. This is called the *value set*. When a valid relation is identified, the *value sets* are compared using an algorithm similar to the one shown in Algorithm 4. This happens during the causality mining process between log formats described above.

Finally, using information gathered from the multiple mining phases described above, DMAC generates a dataflow model for the entire system.

IV. Experimental Evaluation of DMAC

This section discusses experimental results from applying DMAC to several open-source projects, and the accuracy of its reconstructed dataflow models.

A. Experimental Setup

We applied DMAC to the system execution traces generated by following open-source projects: Apache ANT (ant.apache.org); Apache Tomcat Web Server (tomcat.apache.org); ActiveMQ Java Messaging Server (JMS) (activemq.apache.org); and the Deployment And Configuration Engine (DAnCE) [15], which is an implementation of the Object Management Group (www.omg.org) Deployment & Configuration (D&C) [16] specification for component-based distributed systems.

The motivation for selecting these projects is because we used them in prior research efforts involving UNITE [17], but we manually constructed their dataflow model. We therefore have "ground truth" for these project's dataflow models. Another reason is that these projects have diverse system execution traces—giving DMAC a range of case studies. For example, Apache Tomcat's system execution trace is dense compared to Apache ANT, Apache ActiveMQ, and DAnCE. DAnCE's system execution trace, however, has less reoccurring patterns compared to ANT and ActiveMQ.

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 functionality of UNITE and DMAC, however, is not bound to any operating system. Finally, CUTS [18] logging facilities are used to collect and store the log messages captured by appenders [19] and interceptors [20] in a relational database for analysis by DMAC.

B. Experimental Results for System Execution Traces without Domain-specific Evidence

1) Experimental Results for ANT: ANT's system execution trace used in this experiment contained 8100 log messages. We used an initial *min-sup* value of 0.8. As shown in Listing 5, DMAC identified 7 log formats, which cover 94.85% of total system execution trace. DMAC was able to correctly identify the static and variable parts in all 7 log formats. It is worth noting that DMAC identified `LF3` although it has occurred only once in the system execution trace. This is because its corresponding candidate sequence is same as that of `LF2`, which occurs frequently. Because of the difference in position vectors, DMAC was able to distinguish between the two different log formats.

We did not use domain-specific evidence for ANT's system execution trace because it has only one execution context. DMAC therefore identified 12 relations as shown in Listing 5. One of the identified relations has a cause-effect relationship between its variables (*i.e.*, $LF4.1 = LF5.1$). This means that variable 1 in `LF4` has a cause-effect relationship with variable 1 in `LF5`. The other log formats do not have such causal relations among variables.

```
LF1 = {} skipped - don't know how to handle it {}
LF2 = {} omitted as {} is up to date. {}
LF3 = omitted as {} is up to date. {}
LF4 = Task {} started. {}
LF5 = Task {} finished. {}
LF6 = adding directory {} {}
LF7 = adding entry {} {}

LF1->LF2; LF1->LF6; LF1->LF7; LF2->LF6; LF2->LF7;
LF2->LF3; LF4->LF5; LF4->LF1;LF4->LF2; LF4->LF7;
LF4->LF6; LF6->LF7

LF4.1 = LF5.1

Total Records = 8100
LF1 - Count = 1783; Percent = 22.0123%
LF2 - Count = 3488; Percent = 43.0617%
LF3 - Count = 1; Percent = 0.0123457%
LF4 - Count = 159; Percent = 1.96296%
LF5 - Count = 158; Percent = 1.95062%
LF6 - Count = 294; Percent = 3.62963%
LF7 - Count = 1800; Percent = 22.2222%
Total coverage : 94.8518%
```

Listing 5. Results for applying DMAC to ANT's system execution trace.

ANT's dataflow model reconstructed by DMAC is useful for analyzing its performance properties. For example, `LF4` and `LF5` can be used to evaluate the execution time of different ANT tasks, which we did in prior work using a manually constructed dataflow model [17]. Table II shows the results of this analysis using the reconstructed dataflow model, which is the same as the results from our prior work using the manually constructed dataflow model.

2) Experimental Results for ActiveMQ: ActiveMQ's system execution trace contained 3650 log messages. Although ActiveMQ's system execution trace contained fewer messages than ANT, it contained more log formats. DMAC's frequent-sequence mining step produced few sequences for relatively

TABLE II
ANALYSIS OF ANT TASK EXECUTION TIMES USING THE DATAFLOW MODEL RECONSTRUCTED BY DMAC

Task	Exec. Time (msec)	Task	Exec. Time (msec)
available	6.0	macrodef	8.0
chmod	21.0	mkdir	2.1
copy	353.5	path	12.0
delete	37.5	patternset	2.1
echo	69.0	property	8.3
filter	2.0	sequential	2141.4

high *min-sup* value. For example, an initial *min-sup* of 0.16 generated only 2 frequent-sequences; whereas *min-sup* of 0.03 produced 6000 frequent-sequences.

In the latter case, executing DMAC for one iteration produced a dataflow model with 12 log formats and 50.96% coverage. DMAC's iterative mining-process improved when starting from a *min-sup* of 0.16 by producing a dataflow model with 80.23% coverage. This dataflow model contained of 21 log formats and 31 relations. Similar to the ANT case study above, we successfully validated that the reconstructed dataflow model could analyze the same performance property of ActiveMQ that we analyzed in prior work [17] using a manually constructed dataflow model.

3) Experimental Results for Apache Tomcat: Tomcat's system execution trace contained 101700 log messages produced by six different threads. We started DMAC with an initial *min-sup* value of 0.05. After 7 iterations, DMAC identified 89 log formats that covered 61.95% of the system execution trace. DMAC also identified 318 relations. We stopped DMAC after 7 iterations because after 7 iterations DMAC started identifying variable parts of the log messages as static parts. Finally, we validated that the dataflow model reconstructed by DMAC could analyze the same performance properties of Tomcat that were analyzed using a manually constructed dataflow model in prior work [17].

4) Measuring the Reconstructed Dataflow Model's Accuracy: For ANT, ActiveMQ, and Tomcat, we were able to use the reconstructed dataflow model to analyze performance properties that were analyzed using a manually constructed dataflow model. Although the reconstructed dataflow models were correct, we need to validate the accuracy of its log formats and relations in relation to the original source code.

We evaluated the accuracy of a log format with respect to (w.r.t.) its counterpart in the original source by comparing the static and variable parts of the log format correspond with static and variable parts in the originating log message from source code. We also evaluated the accuracy of a log format with respect to the system execution trace by evaluating that (1) the static part of a log format is constant in all corresponding log messages; and (2) the empty placeholder ({}) corresponds to at least two possible values in all the corresponding log message instances in the system execution trace.

Table III shows the accuracy results for the Apache, ANT, and ActiveMQ results previously discussed. Most of the inaccuracy presented in Table III is because variable parts were identified as static parts. This has a direct relationship with the structure of the system execution trace. For example, ANT's system execution trace is very succinct; whereas, ActiveMQ and Tomcat's system execution trace is verbose.

TABLE III
ACCURACY OF AUTO-CONSTRUCTED LOG FORMATS.

Item	ANT	ActiveMQ	Tomcat
# of identified log formats	7	21	89
# of identified log formats checked for correctness	7	21	25
LFs correct w.r.t. source	7	14	18
LFs correct w.r.t. execution trace	6	15	20
source code accuracy	100%	66.7%	72%
execution trace accuracy	85.7%	71.4%	80%

We evaluated the dataflow model's relation accuracy similar to how we evaluated the log format's accuracy. More specifically, we evaluated $(LF_i \rightarrow LF_j)$ accuracy by comparing whether they actually occur in the original source code. When the relation is from the same execution context we check whether the two log formats are generated from adjacent log statements in the source code. When the relations are in different execution contexts we check whether the two log statements represent initiation or completion of a remote procedure call in the source code.

We were able to conclude that the identified relations are 100% accurate when the log formats are from the same execution context. The next section describes an experiment that was conducted to evaluate relation accuracy when the two log formats are from different execution contexts.

C. Experimental Results for System Execution Traces with Domain-specific Evidence

The DAnCE system execution trace was small compared to that of Tomcat, ANT and ActiveMQ. Most of the words in the DAnCE's log messages that describe an event have a low frequency value. Furthermore, the frequency values of log message metadata, *e.g.*, log message severity, was greater than the frequency values for actual log message content. DMAC therefore interprets the metadata as static parts and the remaining content of the actual message as variable parts.

```
[LM_TRACE] -- plan,  [LM_TRACE] -- for,
[LM_TRACE] -- instance,  [LM_TRACE] -- plugin,
-- artifact,[LM_TRACE] -- from,  [LM_TRACE],
-- installation, [LM_TRACE] -- successfully,
[LM_TRACE], -- to, -- handler
```

Listing 6. Frequent sequences for DAnCE's system execution trace.

For example, Listing 6 shows frequent-sequences identified by DMAC when the intermediate *min-sup* is 0.17. These candidate sequences are not sufficient to build a log format. Because of this DMAC considers remaining parts of the log format as variable parts. Listing 7 shows some of the log formats identified by DMAC for DAnCE. From this experiment, we concluded that DMAC does not work well with this system execution trace because it does not have a high frequency

value for the words that describe an event when compared to other parts of the log message. Moreover, it is harder to reconstruct DAnCE's dataflow model since many of the log formats occur in different execution contexts—unlike our previous experiments.

```
LF1 = {} [LM_TRACE] − {} − {} {} {} − {} {} {}
for name {}
LF2 = {} [LM_TRACE] − {} − {} − {} {} {}
for name {}
LF3 = {} [LM_TRACE] − {} − {} − {} {}
for name {} {} {} {}
LF4 = {} [LM_TRACE] − {} − {} {} {} − {} {}
for name {} {} {} {}
```

Listing 7. Some of the log formats identified by DMAC for DAnCE's system execution trace.

Because we were not able to auto-construct a valid dataflow model from DAnCE using only time and variable evidence, we added domain-specific evidence to the auto-construction process. Based on our domain-knowledge of DAnCE, we defined domain-specific evidence at the DMAC-level such that we specified a uniform *bba* value for each causal relation between log formats. More specifically, if our confidence about the domain knowledge is 0.8, then we assigned a *bba* value of 0.8 for each relation $LF_i \rightarrow LF_j$, *i.e.*, $(\{(LF_i \rightarrow LF_j) = YES\}, 0.8)$. Likewise, we assigned a *bba* value of 0.8 for the hypothesis $\{(LF_i \rightarrow LF_k) = NO\}$ for any relation we knew could not occur in the dataflow model.

We then used DMAC with the added domain-specific evidence to auto-construct the dataflow model for DAnCE. Because the domain-specific evidence is designed to produce more accurate results, we evaluated the effect of domain-specific evidence on impacting *true-positives (TP)* and *false-positives (FP)* in the auto-construction process. Figure 4 shows

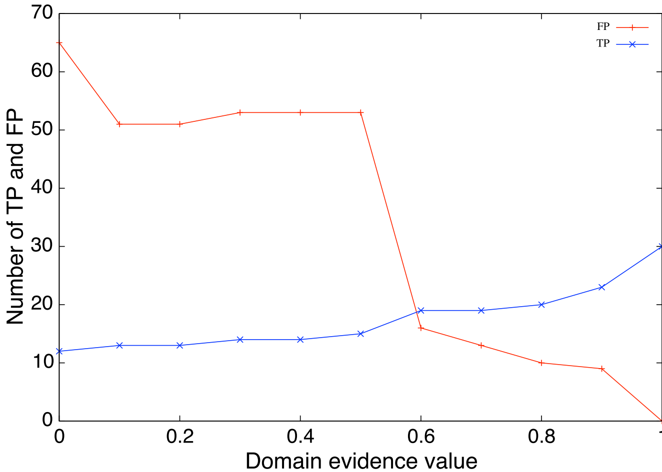

Fig. 4. Effect on domain-knowledge on TPs and FPs in the dataflow model auto-construction process.

the effect that the confidence level of domain-specific evidence has on TPs and FPs. As shown in this figure, as we increase our confidence level, the number of TPs increases and the number of FPs decrease. Likewise, when we reduce our confidence level, the opposite occurs. For example, when the confidence level is 1 (*i.e.*, the tester has complete knowledge and confidence), then DMAC produces the most accurate results. Likewise, a confidence level of 0 produces results similar only using time and variable evidence in the auto-construction process.

The DAnCE results show two evidences (*i.e.*, time and variable) are not always enough to correctly auto-construct a dataflow model. In some cases, it may be necessary to integrate domain-specific evidence. As illustrated in the DAnCE results, DMAC is able to successfully integrate domain-specific evidence into the dataflow model auto-construction process. It is therefore the responsibility of the tester to identify the domain-specific evidence and quantify it correctly to reduce the number of FPs.

V. Related Work

Lou et al. [14] work on identifying dependences among events in distributed systems is most closely related to our work on DMAC. In their work, they use a learning approach based on Bayesian decision model to construct a dependency graph using abstract log formats. Moreover, their Bayesian decision model uses both time and variable evidences. Our work on DMAC extends their work in two ways. First, DMAC uses DS theory, which does not require any upfront training. Secondly, DMAC supports domain-specific evidences, which allows testers to construct more accurate models when both time and variable evidence is not enough.

Qiang et al. [21] describes a technique to detect execution anomalies in distributed systems using unstructured log analysis. Similar to log formats, Qiang et al. has a concept called *log keys*. They also consider the variable parts of log keys as parameters. Unlike DMAC, Qiang et al. uses empirical rules to identify log key parameters. Their intuition is that in log messages, log key parameters are often in the forms of numbers, URIs, and IP addresses. DMAC relaxes this assumption for variable parts of a log message.

Xu et al. [2] describes a technique for mining console logs for the large-scale software system problem detection. Similar to DMAC, Xu et al. extracts static and variable parts of a log message. Their approach, however, requires the original source code combined with the console log. DMAC does not require the original source code to identify static and variable parts of a log message.

Nagappan et al. [11] describes a technique to find log formats without the original source code. DMAC's approach for identifying log formats is similar to the technique described by Nagappan. The main drawback with their approach is variable parts are not identified when they have a frequency above the threshold frequency. DMAC addresses this limitation by mining the system execution trace in iterations where system execution trace is pruned in each iteration based on the identified log format—allowing infrequent log messages to become frequent.

Safyallah et al. [22] uses frequent-sequence mining to identify common functionality associated with feature-specific task scenarios based on data captured in a system execution trace. Safyallah et al. also uses frequent-sequence mining to

discover relationships between different feature specific tasks in inter/intra modules. Their approach defines a frequent-sequence pattern (*i.e.*, execution pattern) as a contiguous part of an execution trace. This, however, cannot be assumed with DMAC because DMAC must account for variable parts appearing in between static parts.

VI. Concluding Remarks

It is hard to analyze system execution traces of a distributed system without a proper dataflow model. Moreover, it is hard to define a dataflow model as the system increases in both size and complexity. This paper therefore presented a tool and method called DMAC that can assist in defining dataflow models used to analyze system execution traces via UNITE. This way, distributed system testers can focus more on analyzing QoS properties as opposed to managing low-level testing artifacts (*e.g.*, the dataflow model). Based on experience gained from applying DMAC to several open-source applications, we have learned the following lessons:

- **Initial *min-sup* value is important.** As shown in the results, the value of *min-sup* has a considerable impact on the quality of the results. For example, lower the *min-sup* value, the more log formats produced. This, however, has the side-effect of incorrectly identifying variable parts of the log format as static parts. DMAC is able to reduce this side-effect using an iterative mining process where initial iterations have a higher *min-sup* value compared to the later ones. This process also improves the coverage.
- **Domain-knowledge is important when construct dataflow models.** We observed that causality mining should not be dependent on only time and variable evidences. This is because the absence—or presence—of either evidence does not necessarily imply two log formats are—or are not—causally related. Time and variable evidences only increases/decreases confidence levels about about the causal relationships. As discussed in this paper, domain-specific evidence played an important role in reconstructing dataflow models where time and variable evidence is not enough. Future work therefore includes developing different techniques for interpreting the user-level domain-specific evidence into DMAC-level domain-specific evidence so it can be better integrated into the dataflow model auto-construction process.
- **Stopping criteria for the iterative mining process is important.** When to stop the iterative mining process is important because DMAC finds more log formats as the number of iterations increases. This, however, may lead to incorrectly identifying variable parts as static parts. In some cases, the *min-sup* value does not decrease during sub-sequent iterations, which is a good indicator of coverage for the auto-generated dataflow model. Future work therefore includes investigating improved techniques for stopping the iterative mining process.

DMAC is integrated into the CUTS system execution modeling tool and is freely available from the following location: cuts.cs.iupui.edu.

References

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

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

[3] M. Nagappan, K. Wu, and M. A. Vouk, "Efficiently extracting operational profiles from execution logs using suffix arrays," in *Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*, ser. ISSRE'09, 2009, pp. 41–50.

[4] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz, "Analysis of a Very Large Web Search Engine Query Log," *SIGIR Forum*, vol. 33, pp. 6–12, September 1999.

[5] 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, pp. 406–415.

[6] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Data Engineering, 1995. Proceedings of the Eleventh International Conference on.* IEEE, 1995, pp. 3–14.

[7] G. Shafer, *A mathematical theory of evidence*. Princeton university press Princeton, 1976, vol. 76.

[8] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, 1996, pp. 3–17.

[9] M. J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning Journal*, vol. 42, no. 1/2, pp. 31–60, Jan/Feb 2001.

[10] P. Tzvetkov, X. Yan, and J. Han, "Tsp: Mining top-k closed sequential patterns," *Knowledge and Information Systems*, vol. 7, no. 4, pp. 438–457, 2005.

[11] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, 2010, pp. 114–117.

[12] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[13] L. Mazlack, "Causality recognition for data mining in an inherently ill defined world," in *International Joint Workshop on Soft Computing for Internet and Bioinformatics*, 2003.

[14] J. Lou, Q. Fu, Y. Wang, and J. Li, "Mining dependency in distributed systems through unstructured logs analysis," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 91–96, 2010.

[15] 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.

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

[17] T. M. Peiris and J. H. Hill, "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, April 2012, pp. 162–171.

[18] J. H. Hill, D. C. Schmidt, J. Edmondson, and A. Gokhale, "Tools for Continuously Evaluating Distributed System Qualities," *IEEE Software*, July/August 2010.

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

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

[21] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," in *IEEE International Conference on Data Mining*, 2009, pp. 149–158.

[22] 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.