Mining Temporal Invariants from Partially Ordered Logs

Ivan Beschastnikh Yuriy Brun Michael D. Ernst Arvind Krishnamurthy Thomas E. Anderson

Computer Science & Engineering
University of Washington

Abstract

A common assumption made in log analysis research is that the underlying log is totally ordered. For concurrent systems, this assumption constrains the generated log to either exclude concurrency altogether, or to capture a particular interleaving of concurrent events. This paper argues that capturing concurrency as a partial order is useful and often indispensable for answering important questions about concurrent systems. To this end, we motivate a family of event ordering invariants over partially ordered event traces, give three algorithms for mining these invariants from logs, and evaluate their scalability on simulated distributed system logs.

1. Introduction

Most log analysis research and tools consider totally ordered (TO) logs of events. A total order has important advantages, such as the ability of humans to directly inspect and understand the logged information and the applicability of simple and powerful log analysis techniques. Further, many well-established logging formats are TO.

However, in domains with concurrent executions, such as distributed systems, events are partially ordered (PO) rather than TO. Concurrency is increasingly used for improving performance on clusters and multi-core architectures. Even simple applications are becoming concurrent, with more functionality migrating into the cloud and with widespread use of Ajax to mask latency.

Because of the widespread concurrency of modern software, we propose that log analysis researchers should take up the challenges and opportunities offered by PO logs. This paper focuses on such logs in the context of distributed systems, which are inherently concurrent.

A PO log has more concurrency information than a TO log, making it more useful when studying concurrent system behavior. For example, consider a simple distributed system with two communicating processes. One of the processes generates event *a*, and the other process generates event *b*. If the events are logged in a TO log (say, a log in which the order of events in the log file implicitly defines the ordering), then it is unclear whether *a* necessarily preceded *b* or just happened to be logged before *b*. Even if *a* and *b* appear in different orders in different traces, it is still possible that they can never occur concurrently. Unlike a TO log, a PO log makes concurrency explicit: the log indicates whether *a* and *b* have some order — in which case they are not concurrent — or they are not ordered — in which case they are concurrent.

A PO log contains more information, and is therefore more complex, than a TO log. It is often infeasible to analyze it manually. Prior work on automated PO log analysis has concentrated on visualization to simplify analysis [5, 24]. In this paper, we define a set of event-ordering invariant templates that capture interesting patterns in PO logs. We then describe algorithms for efficiently mining these invariants from PO logs. In addition to helping developers better understand their systems and detect erroneous behavior, the mined invariants can also be used to infer higher-level system properties of interest to developers. For example, invariants could be combined with performance information to determine whether there is correlation between slow executions and the ordering of events in the system. As well, invariants can help diagnose distributed deadlock, which typically occurs because of an anomalous ordering of distributed events.

We have implemented the algorithms described in this paper and integrated them with Synoptic [3], our open-source, log-analysis tool that is available for download at `http://synoptic.googlecode.com`.

The next section describes an example PO log to introduce the concepts that Section 3 defines more formally. Section 4 formalizes invariants and presents three mining algorithms for extracting invariants from PO logs. Section 5 evaluates the algorithms' performance and scalability. Section 6 discusses future work, while Section 7 overviews related work. Finally, Section 8 concludes the paper.

2. Motivating example

As a motivating example, consider the PO log in Figure 1a. This log captures five executions of a web application in which two clients access a server to buy airplane tickets. Unfortunately, there is only one ticket available. The log captures several scenarios in which the clients and the server interact: different orders of checking for ticket availability, attempts to buy a ticket, and so on.

It is difficult to piece together the various behaviors just by looking at a PO log. Even if one considers the corresponding time-space diagrams in Figure 1b the overall behavior of the system remains obscure. However, the system has definite patterns in its logs, and these patterns can be mined and shown to a developer to aid the developer's understanding of how the system behaves.

Figure 1c lists a subset of the invariants that our Synoptic tool mines from the log in Figure 1a. One mined invariant is that the server cannot sell a ticket after it has sold out of tickets. That is, the *sold-out* server event is never followed by the *sold* server event (sold-out$_s \not\rightarrow$ sold$_s$ in Figure 1c). This temporal property helps to elucidate the server's operation, but it is also simple enough to find and check manually by considering all the server's timelines in the

SLAML'11, October 23, 2011, Cascais, Portugal.

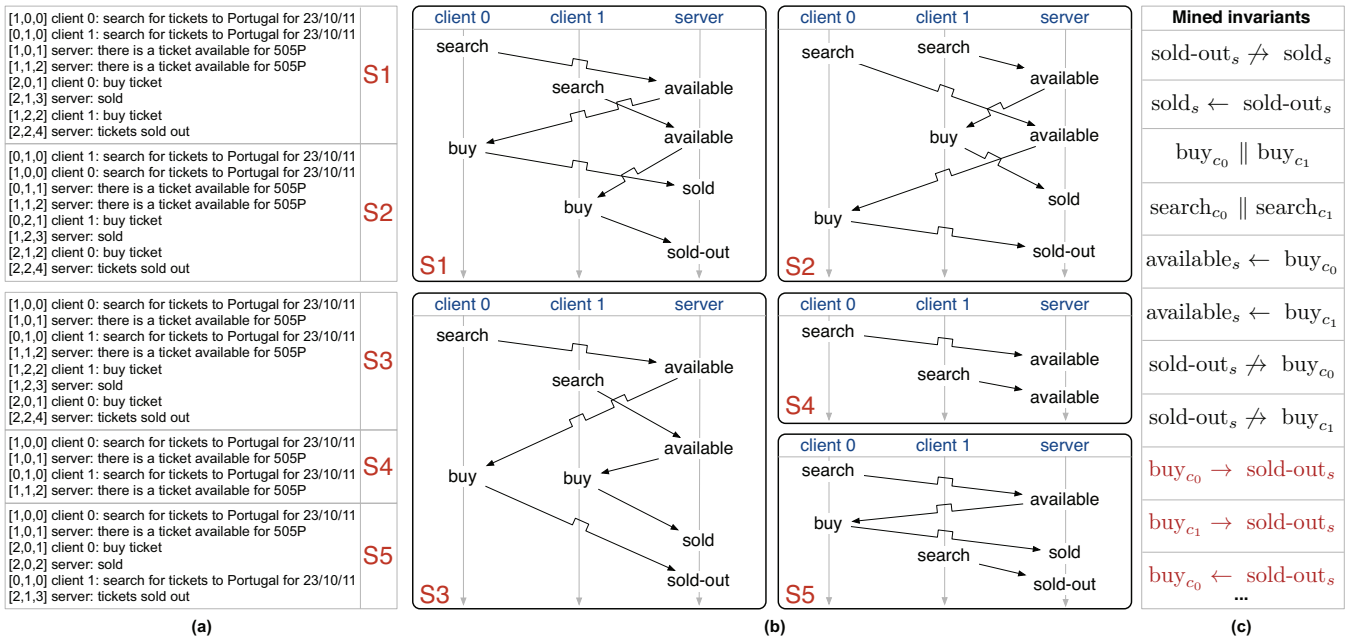

Figure 1: **(a)** Five system traces (S1–S5) for a web application that sells airplane tickets. Each trace consists of log lines and corresponding vector clock timestamps. In the traces, two clients access a single server. **(b)** A visualization of the five system traces as space-time diagrams. Time flows down, and events at each host are shown in a single column. **(c)** A *partial* list of ordering invariants mined from the traces (c_i stands for client i, and s stands for the server). The three invariants at the bottom (in red) are examples of false positives.

time space diagrams in Figure 1b. A less apparent mined invariant is that the clients' *buy* operations are always concurrent ($\text{buy}_{c_0} \parallel \text{buy}_{c_1}$ in Figure 1c). That is, if both clients issue a *buy* command, then a *buy* at client 0 is always concurrent with a *buy* at client 1.

To verify that the system is working properly, a developer can mine the invariants from the log of the captured system executions and verify that the mined invariants are consistent with the developer's understanding of how the system is supposed to behave. The more complex ordering invariants over PO logs allow developers to do this for concurrent systems, such as the example ticket selling system. It is possible, however, that the mining algorithm reports an invariant that is not intended to be true of the system (e.g., the three bottom invariants in Figure 1c) but is only true of the observed executions, such as those induced by a test suite. In this case, the developer knows to write more diverse test cases to exercise more system behavior.

Before explaining invariants in more detail, we first provide a few formal definitions, and explain the vector clock timestamps shown in the example log (in the left column of Figure 1a).

3. Definitions

We assume a distributed system that is composed of h hosts, indexed from 0 to $h-1$. Each host generates a TO sequence of event instances, each of which has an event type from a finite alphabet of host event types.

Definition 1 (Host event types)**.** For all hosts i, the *host event types* set $E_i \supseteq \{START_i, END_i\}$ is a finite set (alphabet) of event types that can be generated by host i.

Definition 2 (System event types)**.** The set E of all possible *system event types* is $\cup E_i$, for all hosts i.

Definition 3 (Event instance)**.** An *event instance* is the triple $t = \langle e, i, k \rangle$, where $e \in E_i$, i is a host index, and k is a non-negative integer that indicates the order of the event instance, among all event instances on host i.

A host trace (Definition 4) is the set of all event instances generated at host i. This includes event instances of type $START_i$, and END_i, which respectively start and end the host trace.

As the example in Figure 1 illustrates, order is an important property of a distributed execution. Event instances are ordered in two ways. First, the host ordering (Definition 5) orders any two event instances at the same host. Second, the interaction ordering (Definition 6) orders dependent event instances at different hosts. For example, if hosts use message passing to communicate, a *send message* event instance is ordered before a *receive message* event instance for the same message. Both orderings are partial orderings, otherwise the distributed system can be more simply modeled as a serial execution.

Definition 4 (Host trace)**.** For all hosts i, a *host trace* is a set T_i of all event instances $t = \langle e, i, k \rangle$, such that an event of type e was the k^{th} event generated on host i. The host trace T_i includes two event instances $\langle START, i, 0 \rangle$ and $\langle END, i, n \rangle$, such that n is the largest k for all $t \in T_i$. The k induces a total ordering of event instances in T_i. We denote this total ordering as $<_i$. More formally, $\forall \hat{e_1} = \langle e_1, i, k_1 \rangle \in T_i, \hat{e_2} = \langle e_2, i, k_2 \rangle \in T_i, (\hat{e_1} <_i \hat{e_2} \iff k_1 < k_2)$.

Definition 5 (Host ordering)**.** A *host ordering* $\prec_{host}$ is the union $\cup <_i$.

Definition 6 (Interaction ordering)**.** An *interaction ordering* $\prec_{interact}$ orders pairs of event instances $\langle e_1, i, k_1 \rangle$ and $\langle e_2, j, k_2 \rangle$ such that $i \neq j$.

A system trace is the union of a set of host traces (one per host), which corresponds to a single execution of the system. This union respects the host ordering and the interaction ordering.

Definition 7 (System trace)**.** A *system trace* is the pair $S = \langle T, \prec \rangle$, where $T = \cup T_i$, and $\prec \; = \; \prec_{host} \cup \prec_{interact}$.

Definition 8 (Log)**.** A *log* L is a set of system traces.

A common way of ordering event instances in a system trace is to associate a vector timestamp with each event instance. These

timestamps make the partial order, $\prec$, of event instances in the system trace explicit. We now explain the algorithm using which the vector timestamps are maintained. This explanation corresponds to a system that uses message passing, though vector timestamps can be used for ordering event instances in a system that uses other mechanisms for inter-host communication, such as shared memory.

3.1 Ordering events with vector time

Vector time [9, 20] is a logical clock mechanism that provides a partial ordering of event instances. In a distributed system of h hosts, each host maintains an array of clocks $C = [c_0, c_1, \ldots, c_{h-1}]$, in which a clock value c_j records the local host's knowledge of (equivalently, dependence on) the local logical time at host j. We denote a timestamp's C clock value for host j as $C[j]$.

The hosts update their clocks to reflect the actual ordering of event instances in the system with the following three rules:

1. All hosts start with an initial vector clock value of $[0, \ldots, 0]$.
2. When a host i generates an event instance, it increments its own clock value (at index i) by 1, i.e., $C_i[i]{+}{+}$.
3. When a host h communicates with a host h', h shares its current clock C_h with h', and h' updates its local clock $C_{h'}$ so that $\forall i, C_{h'}[i] = \max(C_h[i], C_{h'}[i])$. The host h' also updates its local clock value as in (2), since message receipt is considered an event.

Note that the above procedure assumes that each host knows the set of participating hosts in the system, and that this set does not change over time.

Using the above procedure, each event instance in the system is associated with a *vector timestamp* — the value of C immediately after the event instance occurred. Vector timestamps can be partially ordered with the relation $\prec$, where $C \prec C'$ if and only if each entry of C is less than or equal to the corresponding entry of C', and at least one entry is strictly less. More formally: $C \prec C'$ iff $\forall i, C[i] \leq C'[i]$ and $\exists j, C[j] < C'[j]$. This ordering is partial because some timestamp pairs cannot be ordered (e.g., $[1,2]$ and $[2,1]$).

The vector timestamp ordering allows us to partially order all the event instances in the system. Figure 1b shows five time-space diagrams[1], each of which represents the $\prec$ ordering for each of the system traces in the log in Figure 1a. For example, in system trace S1 a *search* event instance at client 0 has a timestamp of $[1,0,0]$, which immediately precedes the first *available* event instance at the server, timestamped with $[1,0,1]$. The time-space diagram encodes this precedence information as a directed edge between the two events. However, the same *search* event instance at client 0 is not ordered with the *search* event instance at client 1, which has a timestamp of $[0,1,0]$. Correspondingly, there is no path in the time-space diagram between these two event instances.

The next section formally defines a few kinds of event ordering *invariants* that hold across all system executions, and gives an algorithm to infer these from a log.

4. Mining temporal invariants

Prior work [3, 8, 11, 32] has shown that invariants of software systems are useful across a range of application domains. A common feature of all the prior work on invariant mining is that it only considers TO sequences of event instances. In this section, we first extend temporal invariants to PO logs, and then give three algorithms for mining these invariants.

[1]For compactness, the diagrams in Figure 1b bundle message receipt, message processing, and message send events into one event.

4.1 Temporal invariants in PO logs

We consider five temporal ordering invariants that relate pairs of *host event types*. All the invariants are defined in terms of the $\prec$ partial ordering, which was introduced in Definition 7 and operationalized in Section 3.1. Throughout, we use the notation e_i to represent an event $e \in E_i$, and we use $\hat{e}_i$ to represent a corresponding event instance $\langle e, i, k \rangle \in L$.

Definition 9 (Event invariant)**.** Let L be a log, and let a_i and b_j be two host event types whose corresponding event instances, $\hat{a}_i$ and $\hat{b}_j$, appear at least once in some system trace in L. Then, an *event invariant* is a property that relates a_i and b_j in one of the following five ways, and evaluates to true in each of the system traces in L.

$a_i \rightarrow b_j$ **:** An event instance of type a at host i is **always followed by** an event instance of type b at host j. Formally: $\forall \hat{a}_i \; \exists \; \hat{b}_i, \; \hat{a}_i \prec \hat{b}_j$.

$a_i \not\rightarrow b_j$ **:** An event instance of type a at host i is **never followed by** an event instance of type b at host j. Formally: $\forall \hat{a}_i \;\; \nexists \hat{b}_j, \; \hat{a}_i \prec \hat{b}_j$.

$a_i \leftarrow b_j$ **:** An event instance of type a at host i **always precedes** an event instance of type b at host j. Formally: $\forall \hat{b}_j \; \exists \; \hat{a}_i, \; \hat{a}_i \prec \hat{b}_j$.

$a_i \parallel b_j$ **:** An event instance of type a at host i is **always concurrent** with an event instance of type b at host j. Formally: $\forall \hat{a}_i, \hat{b}_j (\hat{a}_i \not\prec \hat{b}_j \wedge \hat{b}_j \not\prec \hat{a}_i)$.

$a_i \nparallel b_j$ **:** An event instance of type a at host i is **never concurrent** with an event instance of type b at host j. Formally: $\forall \hat{a}_i, \hat{b}_j (\hat{a}_i \prec \hat{b}_j \vee \hat{b}_j \prec \hat{a}_i)$.

We omit the *never precedes* invariant because it is equivalent to the $\not\rightarrow$ invariant. The $\rightarrow$, $\not\rightarrow$, and $\leftarrow$ invariants capture particular kinds of ordering dependency. For example, Figure 1c lists the "available$_s \leftarrow$ buy$_{c_0}$" invariant, which means that a client can only make a ticket purchase if the server indicated ticket availability.

The $\parallel$ and $\nparallel$ invariants are more general. The $\parallel$ invariant captures the *lack of* ordering, and $\nparallel$ captures the *presence of some* ordering. An example of a $\parallel$ invariant is "search$_{c_0} \parallel$ search$_{c_1}$" in Figure 1c, which means that ticket search requests from the two clients are never ordered. The $\parallel$ and $\nparallel$ invariants are also commutative — e.g., $a_i \parallel b_j$ iff $b_j \parallel a_i$.

The $\rightarrow$, $\not\rightarrow$, and $\leftarrow$ invariants are analogs of the most frequently observed specification patterns in Dwyer et al. [7], with scope constrained to a trace (i.e., global scope). The translation is not one-to-one: $a_i \rightarrow b_j$ is Dwyer's Existence pattern when a_i is $START_i$, and is otherwise Dwyer's Response pattern. Another example is $\forall b_j, a_i \leftarrow b_j$, which is Dwyer's Universality pattern. A key difference, however, is that Dwyer et al. [7] study specification patterns of sequential, rather than concurrent, systems.

For each of the five invariants, the event instances may occur on different hosts or on the same host. We term invariants that relate host event types on the same host (i.e., $i = j$) as *local*, and those that relate host event types on different hosts ($i \neq j$) as *distributed*. Local invariants can be evaluated independently of event instances on other hosts, solely by using the total ordering of event instances on the host (i.e., $<_i$). In contrast, distributed invariants capture dependency between event types on different hosts — their evaluation requires the use of the partial ordering.

Some invariants are trivially true, such as $START_i \rightarrow END_i$. Others are trivially false, such as $START_i \not\rightarrow END_i$. However, determining the truth value of most invariants requires log analysis, which we take up next.

4.2 Mining temporal invariants

The task of mining invariants involves taking a log (e.g., Figure 1a) as input, and outputting the set of invariants that are true of the log (e.g., Figure 1c). To process the log, we assume that the user also inputs a set of regular expressions so that each line in the input log can be parsed into a vector timestamp and an event instance.

To simplify our discussion of invariant mining algorithms, we use the directed acyclic graph (DAG) representation of a system trace. The time-space diagrams in Figure 1b illustrate the basic idea of the DAG representation (except that in these diagrams, edges between events generated at the same host are implicit). Formally, a system trace $\langle t, \prec \rangle$ can be represented as a DAG with nodes corresponding to event instances in t, and an edge from e to e' iff e is a direct predecessor of e' (i.e., iff $e \prec e'$ and $\nexists\, e''$, $e \prec e'' \prec e'$).

4.2.1 *Transitive-closure-based algorithm*

One invariant-mining algorithm computes the forward and reverse transitive closures of each trace DAG in L and then determines which invariants are valid from those transitive closures as follows:

- $a_i \rightarrow b_j$ iff in each forward DAG transitive closure, every $\hat{a_i}$ node (instance of event type a_i) has an edge to a $\hat{b_j}$ node.
- $a_i \not\rightarrow b_j$ iff in each forward DAG transitive closure, every $\hat{a_i}$ node has no edge to a $\hat{b_j}$ node.
- $a_i \leftarrow b_j$ iff in each reverse DAG transitive closure, every $\hat{b_j}$ node has an edge to a $\hat{a_i}$ node.
- $a_i \parallel b_j$ iff there are no edges between $\hat{a_i}$ and $\hat{b_j}$ nodes in the forward DAG transitive closure, and the two kinds of nodes both occur in some trace DAG.
- $a_i \nparallel b_j$ iff there is an edge between each $\hat{a_i}$ node and some $\hat{b_j}$ node, and vice versa, in the forward DAG transitive closure for DAGs where both nodes occur.

This algorithm performs poorly on sparse DAGs, for which transitive closure construction is expensive. Next, we describe two algorithms that do not explicitly generate the transitive closures, but instead mine invariants implicitly by collecting event type co-occurrence counts.

4.2.2 *Co-occurrence counting algorithm v1*

The idea behind both co-occurrence counting algorithms (v1 and v2) is to avoid explicit construction of the trace DAGs' transitive closures. Instead, the algorithms walk through the trace DAGs and count specific values, such as the number of times an event instance of type a_i is followed by an event instance of type b_j. After counting, the algorithms use a set of rules (derived from the invariant definitions in Section 4.1) to infer the invariants. The first version of the algorithm (v1) uses the following rules:

- $a_i \rightarrow b_j$ iff the number of $\hat{a_i}$ occurrences is equal to the number of times that $\hat{a_i}$ was followed by $\hat{b_j}$.
- $a_i \not\rightarrow b_j$ iff the number of times that $\hat{a_i}$ was followed by $\hat{b_j}$ was 0.
- $a_i \leftarrow b_j$ iff the number of $\hat{b_j}$ occurrences is equal to the number of times $\hat{b_j}$ was preceded by $\hat{a_i}$.
- $a_i \parallel b_j$ iff $\hat{a_i}$ and $\hat{b_j}$ co-occurred at least once in a system trace (otherwise calling the two event types concurrent does not make sense); and the number of times that $\hat{a_i}$ followed $\hat{b_j}$ and the number of times $\hat{b_j}$ followed $\hat{a_i}$ was 0 (the events were never ordered).

```
1   Input: event log L, as a set of event instance DAGs
2
3   for dag ∈ L {
4     let dagOcc[] // Maintains DAG event counts per event type
5     // Traverse the DAG in the forward direction:
6     foreach node ∈ dag, in topological order:
7       let node.predecessors = ∪_{p∈node.parents} p.predecessors
8       let b_j = node.type
9       dagOcc[b_j] ++
10      let seenTypes = {}
11      for nodeP ∈ node.predecessors:
12        let a_i = nodeP.type
13        PrecPairs[a_i][b_j] ++
14        if a_i ∉ seenTypes:
15          CoOcc[a_i][b_j] = true
16          Prec[a_i][b_j] ++
17          seenTypes = seenTypes ∪ {a_i}
18
19    // Traverse the DAG in the reverse direction:
20    foreach node ∈ dag, in reverse topological order:
21      let node.successors = ∪_{c∈node.children} c.successors
22      let a_i = node.type
23      let seenTypes = {}
24      for nodeS ∈ node.successors:
25        let b_j = nodeS.type
26        FollowsPairs[a_i][b_j] ++
27        if b_j ∉ seenTypes:
28          Follows[a_i][b_j] ++
29          seenTypes = seenTypes ∪ {b_j}
30
31    // Accumulate this DAG's event instance counts:
32    for a_i ∈ dagOcc.keys:
33      Occ[a_i] += dagOcc[a_i]
34      for b_j ∈ dagOcc.keys:
35        TraceCountProductsSum[a_i][b_j] +=
36          (dagOcc[a_i] * dagOcc[b_j])
37  }
38
39  // Use the counts to derive the invariants:
40  let invariants = []
41  for a_i, b_j ∈ eventTypes :
42    if Follows[a_i][b_j] = Occ[a_i]:
43      invariants.append(a_i → b_j)
44    if Follows[a_i][b_j] = 0:
45      invariants.append(a_i ↛ b_j)
46    if Prec[a_i][b_j] = Occ[b_j]:
47      invariants.append(a_i ← b_j)
48    if CoOcc[a_i][b_j] ∧ Follows[a_i][b_j] = 0 ∧ Follows[b_j][a] = 0:
49      invariants.append(a_i ∥ b_j)
50    if CoOcc[a_i][b_j] ∧ TraceCountProductsSum[a_i][b_j] =
51      PrecPairs[a_i][b_j] + FollowsPairs[b_j][a_i]:
52      invariants.append(a_i ∦ b_j)
53
54  Output: invariants
```

Figure 2: The co-occurrence counting algorithm v1 described in Section 4.2.2.

```
3   for dag ∈ L {
4     // Traverse the DAG in the forward direction:
5     foreach node ∈ dag, in topological order:
6       let node.typePred = ∪_{p∈node.parents} p.typePred
7       let b_j = node.type
8       Occ[b_j] ++
9       for a_i ∈ node.typePred:
10        CoOcc[a_i][b_j] = true
11        Prec[a_i][b_j] ++
12
13    // Traverse the DAG in the reverse direction:
14    foreach node ∈ dag, in reverse topological order:
15      let node.typeSucc = ∪_{c∈node.children} c.typeSucc
16      let a_i = node.type
17      for b_j ∈ node.typeSucc:
18        Follows[a_i][b_j] ++
19  }
```

Figure 3: A different *for* loop body for the pseudocode in lines 3–29 of Figure 2, which generates a simpler and more efficient algorithm that mines all of the invariant types except $\nparallel$. This algorithm is the co-occurrence counting algorithm v2, described in Section 4.2.3. As well, this algorithm omits lines 50–52 in Figure 2.

- $a_i \nparallel b_j$ iff $\hat{a_i}$ and $\hat{b_j}$ co-occurred at least once in a system trace; and in every trace each $\hat{a_i}$ instance is followed or preceded by every $\hat{b_j}$ instance. That is, in a trace the number of $\hat{a_i}$ followed by $\hat{b_j}$ pairs plus the number of $\hat{a_i}$ preceded by $\hat{b_j}$ pairs must equal the count of $\hat{a_i}$ in the trace times the count of $\hat{b_j}$ in the trace.

The pseudocode in Figure 2 outlines this procedure. The algorithm starts by building a set of data structures to hold the various occurrence counts:

- Occ[a_i] : the count of $\hat{a_i}$ across all traces.
- CoOcc[a_i][b_j] : whether or not $\hat{a_i}$ and $\hat{b_j}$ co-appeared in a trace.
- Prec[a_i][b_j] : the count of $\hat{b_j}$ instances that were preceded by at least one $\hat{a_i}$.
- Follows[a_i][b_j] : the count of $\hat{a_i}$ instances that were followed by at least one $\hat{b_j}$.
- FollowsPairs[a_i][b_j] : the count of all $(\hat{a_i}, \hat{b_j})$ pairs for which $\hat{a_i}$ was followed by $\hat{b_j}$.
- PrecPairs[a_i][b_j] : the count of all $(\hat{a_i}, \hat{b_j})$ pairs for which $\hat{a_i}$ precedes $\hat{b_j}$.
- TraceCountProductsSum[a_i][b_j] : the sum across all traces of the product of the number of $\hat{a_i}$ in a trace and the number of $\hat{b_j}$ in a trace.

To collect these counts, the trace DAG is first traversed in the forward (lines 6–17 in Figure 2) and then in the reverse directions (lines 20–29). Both traversals are in topological order (e.g., on the forward traversal a node is visited after all of its parents). The topological order guarantees that all the nodes that precede (respectively follow) the node's parents (respectively children) are aggregated correctly (lines 32–36). Once the DAG is traversed in both directions, the algorithm infers invariants from the data structures. Each *if* statement on lines 42–52 of the pseudocode infers a single type of invariant and corresponds to one of the informal rules given above.

Because the algorithm traverses each edge once, its base traversal time for a single trace DAG is $\Theta(|E|)$, where E is the set of edges in the DAG. On traversing an edge, the algorithm needs to merge two sets whose sizes are at most $|V|$, where V is the set of nodes in the DAG. Therefore, in processing a single trace DAG, the algorithm has a running time of $\Theta(|E||V|)$. Because it does not need to explicitly maintain a transitive closure, this algorithm performs especially well on sparse trace DAGs.

4.2.3 Co-occurrence counting algorithm v2 (w/o $\nparallel$)

In both of the previous algorithms, the cost of computing the $\nparallel$ invariant is significantly higher than that of computing each of the other invariant types. This is because evaluating the invariant $a_i \nparallel b_j$ requires an algorithm to consider every pair of instances $(\hat{a_i}, \hat{b_j})$. This overhead prompted us to consider an algorithm that mines all of the invariants except the $\nparallel$ invariant.

Figure 3 lists a different *for* loop body for the pseudocode in lines 3–29 of Figure 2. The resulting algorithm — co-occurrence counting algorithm v2 — is significantly faster (see Section 5). The reason for this is that instead of maintaining the set of all *event instances* that precede (respectively follow) a node, the algorithm maintains only the set of *event types* that precede (respectively follow) a node. Because of this, the per-edge cost drops from $\Theta(|V|)$ to $\Theta(|ETypes|)$ where $ETypes$ is the set of event types in the trace DAG. Therefore, this algorithm's running time is $\Theta(|E||ETypes|)$.

5. Evaluation

This section compares the performance of the transitive-closure-based algorithm with the performance of the two co-occurrence counting algorithms. We evaluate the algorithms on synthetic PO logs that we generated using a discrete-time simulator that simulates a set of concurrent communicating hosts. We first describe the simulator, and then present and discuss the results.

5.1 Log-generating system simulator

The simulator is parameterized by the number of hosts, number of events types, number of events per execution, and the number of executions. For each event, the simmulator chooses the host that will execute the event and the event's type, both with uniform probability. The simulator also decides to either associate the event with sending a message to some other random node (with probability 0.3); or, if the node has messages in its queue, to associate the event with receiving a message (with probability 0.4); or to make the event local to the selected host (remaining probability). Any outstanding messages in the receive queues are flushed when the simulation ends.

The simulator maintains vector clocks, following the procedure from Section 3.1. The simulator outputs a log of multiple executions, or system traces, composed of events; each event has a vector timestamp.

5.2 Methodology

We implemented the three invariant mining algorithms in Java and ran experiments on an Intel i7 (2.8 GHz) OS X 10.6.7 machine with 8GB RAM. Our implementation used the Floyd-Warshall [10] algorithm to compute the transitive closure. As part of our future work, we plan to implement a more efficient transitive closure algorithm specific to DAGs (e.g., [14]).

Our evaluation goal was to measure how the two versions of the co-occurrence counting algorithm scale, as compared to the transitive-closure-based algorithm, in four dimensions: (1) with the length of the system trace, (2) with the number of traces in the log, (3) with the number of hosts, and (4) with the number of event types. For each of the dimensions, we first used the simulator to generate a set of logs, varying that dimension and keeping the others constant. The constant values were: 30 hosts, 50 host event types per host (= 1,500 total since event types at different hosts are considered different), 1,000 events per execution, and 50 executions. We ran each algorithm 5 times and report the median value.

5.3 Results from generated logs

Figure 4 plots the results from our experiments. Figure 4(a) illustrates the algorithms' scalability with respect to the length of the system trace and Figure 4(b) with respect to the length of the log (i.e., the number of traces). In both cases, the transitive-closure-based algorithm outperformed the co-occurrence counting algorithm v1. The co-occurrence counting algorithm v2 (without $\nparallel$) performed best.

Figure 4(c) illustrates the algorithms' scalability with respect to the number of hosts and Figure 4(d) with respect to the number of event types. In both cases, the transitive-closure-based algorithm underperformed the co-occurrence counting v1. Again, the co-occurrence counting algorithm v2 performed best.

6. Discussion and future work

To simplify presentation, we omitted certain details about the mining algorithms. For instance, some invariants are logically equivalent, such as $START_i \rightarrow x_i$ and $x_i \leftarrow END_i$. Others, such as the local versions of $\parallel$ and $\nparallel$ are trivial. Also, some invariants may be

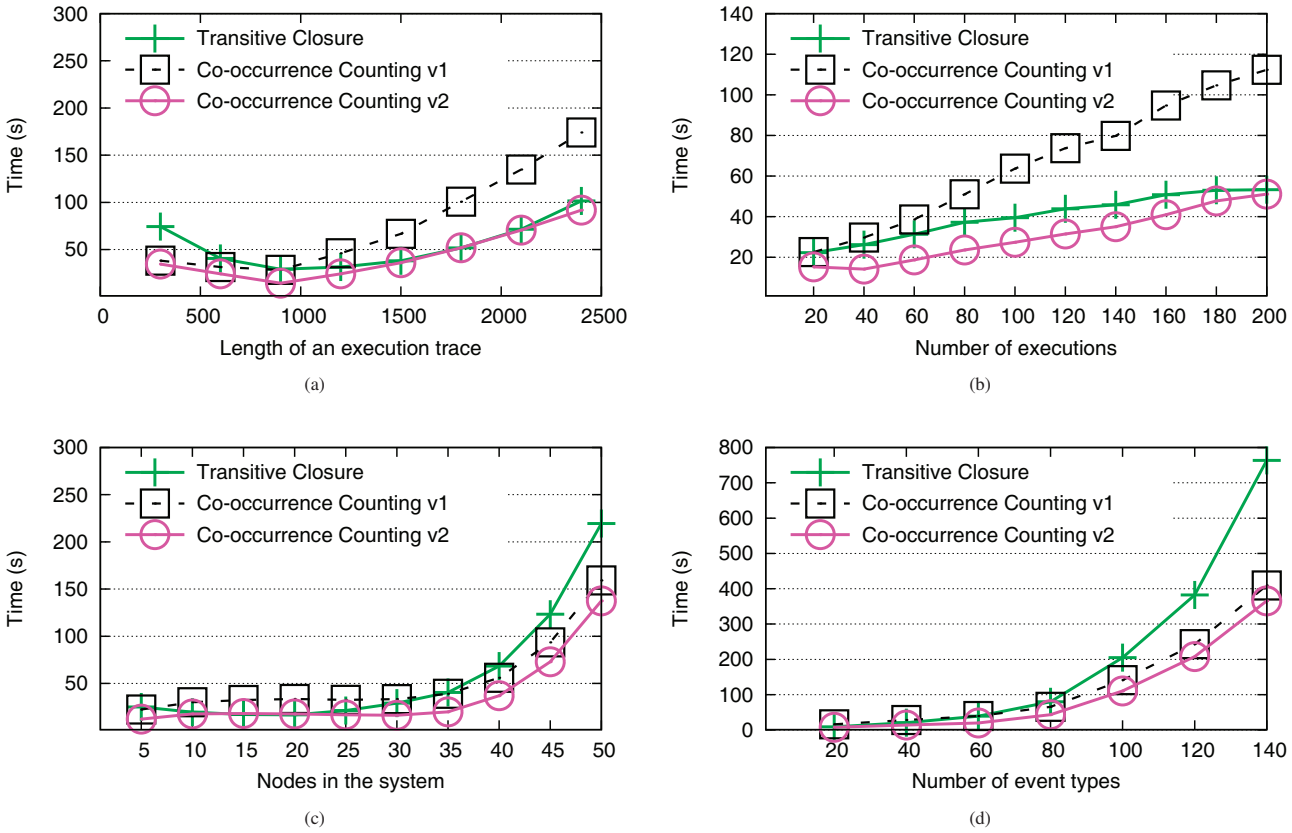

Figure 4: Invariant mining time for the transitive closure and the co-occurrence counting algorithms on logs generated by the simulator described in Section 5.1. In each of the figures, a single log feature is varied: (a) number of hosts, (b) execution length, (c) number of executions, and (d) number of event types. The other log features were held constant in the same figure, and were identical across figures: 30 hosts, 50 host event types per host (= 1,500 total since event types at different hosts are considered different), 1,000 events per execution, and 50 executions.

subsumed by others. For example, the distributed versions of $\rightarrow$ and $\leftarrow$ invariants make stronger claims about event ordering and therefore subsume distributed $\nparallel$ invariants. The mining algorithm implementations detect such duplicate, trivial, and subsumed invariants and filter them out.

We have implemented and compared three invariant mining algorithms that pre-process the log into DAGs to mine invariants. However, it is also possible to mine invariants directly from a log L by first enumerating all the possible invariants based on event types in L, and then traversing each of the system traces in L and checking each invariant to eliminate the false invariants. Algorithm developed by Sen et al. [23] for efficiently checking certain kinds of temporal predicates over consistent cuts of a distributed execution could be used for this. However, efficient traversal of the traces without an explicit DAG structure is non-trivial. We plan to implement this more direct algorithm in our future work. More generally, other approaches such as those based on graph reachability could be used for mining invariants [4]. Counting seems to capture the minimum information necessary for our invariant types, but we want to explore other approaches in our future work.

Also, our evaluation concentrated on mining scalability since system size and log size are a major concern for log analysis in practice. However, we did not evaluate the fundamental assumption that PO log invariants are useful to developers. Although invariants mined from example logs we've considered, like the one given in Section 2, have been helpful to us for improving our own intuition about the logs, further study is necessary. We plan to evaluate the utility of mined invariants and the relevancy of our invariant types with a case study in future work.

We assume the availability of logs, annotated with vector timestamps. A drawback to using vector timestamps in large systems is their performance penalty — vector length scales linearly with the number of hosts in the system and exchanging them may negatively impact network performance. Though more efficient vector clock mechanisms exist [2], we believe that their application can be made practical by limiting their use to short time periods on large systems, or by using vector clocks exclusively for debugging and during development and testing.

7. Related Work

In this section, we summarize three areas of research relevant to ours: invariant mining, distributed system debugging via log analysis, and other types of distributed system debugging techniques.

7.1 Invariant mining

Javert [11] is a specification mining tool that infers complex specifications by composing simpler patterns into larger ones. Javert's invariants are more complex than ours (e.g., it handles invariants over three events). Similarly, Perracotta [32] mines and visualizes temporal properties of event traces. These invariants have been used to study program evolution [31]. All of these systems require TO logs (or observed executions), whereas our work concentrates on PO logs, common in distributed systems.

Daikon [8] observes system executions and mines data structure

invariants as method pre- and post-conditions. Our work concentrates on temporal invariants.

Jiang et al. [17] proposed approximately mining certain types of invariants that relate flow intensities (e.g., traffic volume) in distributed systems. These invariants capture non-temporal properties. In contrast, our proposal is exact, not approximate, and captures temporal properties. Yabandeh et al. [29] describe Avenger, which mines invariants that hold most of the time. These almost-invariants are helpful for finding bugs that manifest infrequently. Avenger mines a rich set of data invariant types; it does not mine temporal properties.

Finally, Lou et al. [19] define a set of event dependencies that range over events in interleaved traces of independent processes. These include what they term *forward* and *backward* dependencies. Our temporal invariants consider communicating processes, as opposed to dependent ones.

7.2 Debugging distributed systems via log analysis

One area in which log analysis is helpful is debugging. Bugs can manifest themselves via anomalous executions. Detecting anomalies in distributed systems is a popular research area [28, 16, 33]. The aim of our invariants is broader: to aid understanding efforts. However, our invariants can also be used for debugging. In fact, temporal invariants mined from systems that generate TO logs have been shown to be helpful for debugging and understanding [3].

Bates et al. [1] developed an event definition language that caused programs to generate logs with deep semantics information, such as hierarchical relationships between events. Their approach requires access to the source code. In contrast, our approach does not need access to the source code, and works on the already generated logs. We do require the developer to express a set of regular expressions. However, this set may be mined automatically [27, 34].

MapReduce-specific research — SALSA [25] and Mochi [26] — has created visualizations helpful to performance debugging of Hadoop [15] node logs. Our approach, of course, is generic and applicable to a wide range of systems.

7.3 Other types of distributed systems debugging

Quality specification can aid debugging, but the process of specifying systems with tools like CADP [12] is difficult and has not gained wide adoption by system builders. Automatically mined invariants can serve as partial specifications and can be used to compare the system's implementation to the developers' understanding of the system.

A set of mined invariants can also be leveraged by tools, such as runtime checkers, which ensure that the system conforms to the developer's expectations. Violated invariants can be reported to the developer [22, 13, 18, 6], or the system may automatically attempt to steer away from the violation [30, 21]. Many existing distributed system debugging tools assume that the developer is willing to write down a set of specifications. Invariants mined from a log, as described in this paper, can help to relieve this specification burden and make prior work that relies on specifications more practical.

8. Conclusion

Mining invariants from totally ordered logs has proven helpful for system debugging and understanding (e.g., [3, 8, 11, 32]. In this paper, we proposed to extend invariant mining to partially ordered logs, which are common in the distributed systems setting. We formally defined relevant invariants for PO logs and described efficient algorithms for mining such invariants. The resulting tool is available for download at `http://synoptic.googlecode.com`

Acknowledgments

This work is supported by the National Science Foundation under grant CNS-0963754 and under grant #0937060 to the Computing Research Association for the CIFellows Project. We also thank the SLAML'11 reviewers and attendees for their constructive feedback.

References

[1] BATES, P. High-level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *Journal of Systems and Software 3*, 4 (Dec. 1983), 255–264.

[2] BECKER, D., RABENSEIFNER, R., WOLF, F., AND LINFORD, J. C. Scalable Timestamp Synchronization for Event Traces of Message-Passing Applications. *Parallel Comput. 35* (Dec. 2009), 595–607.

[3] BESCHASTNIKH, I., BRUN, Y., SCHNEIDER, S., SLOAN, M., AND ERNST, M. D. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the the 8th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2011), ESEC/FSE '11.

[4] CHEN, Y., AND CHEN, Y. An Efficient Algorithm for Answering Graph Reachability Queries. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 893–902.

[5] CONSENS, M. C., HASAN, M. Z., AND MENDELZON, A. O. Debugging Distributed Programs by Visualizing and Querying Event Traces. In *Applications of Databases, First International Conference, ADB-94* (1993), vol. 819, pp. 181–183.

[6] DAO, D., ALBRECHT, J., KILLIAN, C., AND VAHDAT, A. Live Debugging of Distributed Systems. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (Berlin, Heidelberg, 2009), CC '09, Springer-Verlag, pp. 94–108.

[7] DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, 1999), ICSE '99, ACM, pp. 411–420.

[8] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering 27*, 2 (Feb. 2001), 99–123.

[9] FIDGE, C. J. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference* (University of Queensland, Australia, 1988), pp. 55–66.

[10] FLOYD, R. W. Algorithm 97: Shortest path. *Commun. ACM 5*, 6 (June 1962), 345+.

[11] GABEL, M., AND SU, Z. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (New York, NY, USA, 2008), SIGSOFT '08/FSE-16, ACM, pp. 339–349.

[12] GARAVEL, H., LANG, F., MATEESCU, R., AND SERWE, W. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Tools and Algorithms for the Construction and Analysis of Systems*, P. Abdulla and K. Leino, Eds., vol. 6605 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2011, ch. 33, pp. 372–387.

[13] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global Comprehension for Distributed Replay. In *Networked Systems Design and Implementation (NSDI)* (2007).

[14] GORALČÍKOVÁ, A., AND KOUBEK, V. A Reduct-and-Closure Algorithm for Graphs. In *Mathematical Foundations of Computer Science 1979*, J. Becvár, Ed., vol. 74 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1979, pp. 301–307.

[15] Welcome to Apache Hadoop! `http://hadoop.apache.org/`. Accessed April 6, 2011.

[16] JIANG, G., CHEN, H., UNGUREANU, C., AND YOSHIHIRA, K. Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. In *Proceedings of the Second International Conference on Automatic Computing* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 111–122.

[17] JIANG, G., CHEN, H., AND YOSHIHIRA, K. Efficient and Scalable Algorithms for Inferring Likely Invariants in Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering 19*, 11 (Nov. 2007), 1508–1523.

[18] LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J., WU, M., KAASHOEK, M. F., AND ZHANG, Z. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 423–437.

[19] LOU, J. G., FU, Q., YANG, S., LI, J., AND WU, B. Mining Program Workflow from Interleaved Traces. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2010), KDD '10, ACM, pp. 613–622.

[20] MATTERN, F. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms* (1989), pp. 215–226.

[21] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. Automatically Patching Errors in Deployed Software. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Big Sky, MT, USA, Oct. 2009), pp. 87–102.

[22] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3* (Berkeley, CA, USA, 2006), NSDI'06, USENIX Association, p. 9.

[23] SEN, A., AND GARG, V. K. Detecting Temporal Logic Predicates on the Happened-Before Model. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2002), IPDPS '02, IEEE Computer Society.

[24] STONE, J. M. A Graphical Representation of Concurrent Processes. *SIGPLAN Not. 24* (Nov. 1988), 226–235.

[25] TAN, J., PAN, X., KAVULYA, S., GANDHI, R., AND NARASIMHAN, P. SALSA: Analyzing Logs as State Machines. In *Proceedings of the First USENIX conference on Analysis of system logs* (Berkeley, CA, USA, 2008), WASL'08, USENIX Association, p. 6.

[26] TAN, J., PAN, X., KAVULYA, S., GANDHI, R., AND NARASIMHAN, P. Mochi: Visual Log-analysis based Tools for Debugging Hadoop. In *Proceedings of the 2009 conference on Hot topics in cloud computing* (Berkeley, CA, USA, 2009), HotCloud'09, USENIX Association, p. 18.

[27] VAARANDI, R. A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs. In *Proceedings of the 2004 IFIP International Conference on Intelligence in Communication Systems* (2004), vol. 3283, pp. 293–308.

[28] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 117–132.

[29] YABANDEH, M., ANAND, A., CANINI, M., AND KOSTIĆ, D. Finding Almost-Invariants in Distributed Systems. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems* (Oct. 2011).

[30] YABANDEH, M., KNEZEVIC, N., KOSTIĆ, D., AND KUNCAK, V. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2009), USENIX Association, pp. 229–244.

[31] YANG, J., AND EVANS, D. Automatically Inferring Temporal Properties for Program Evolution. In *Proceedings of the 15th International Symposium on Software Reliability Engineering* (Washington, DC, USA, Nov. 2004), IEEE Computer Society, pp. 340–351.

[32] YANG, J., AND EVANS, D. Dynamically Inferring Temporal Properties. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2004), PASTE '04, ACM, pp. 23–28.

[33] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. *SIGPLAN Not. 45* (Mar. 2010), 143–154.

[34] ZHU, K. Q., FISHER, K., AND WALKER, D. Incremental Learning of System Log Formats. *SIGOPS Oper. Syst. Rev. 44* (Mar. 2010), 85–90.