Synoptic: Summarizing System Logs with Refinement

Sigurd Schneider* Ivan Beschastnikh† Slava Chernyak‡ Michael D. Ernst† Yuriy Brun†

*Computer Science, Saarland University †Computer Science & Engineering, University of Washington ‡Google, Inc.

sigurd@ps.uni-saarland.de, {ivan, chernyak, mernst, brun}@cs.washington.edu

Abstract

Distributed systems are often difficult to debug and understand. A typical way of gaining insight into system behavior is by inspecting execution logs. However, manual inspection of logs is an arduous process. To support this task we developed *Synoptic*. Synoptic outputs a concise graph representation of logged events that captures temporal invariants mined from the log.

We applied Synoptic to synthetic and real distributed system logs and found that it augmented a distributed system designer's understanding of system behavior with reasonable overhead for an offline analysis tool. In contrast to prior approaches, Synoptic uses a combination of refinement and coarsening to explore the space of representations. Additionally, it infers temporal event invariants to capture distributed system semantics. These invariants drive the exploration process and are satisfied by the final representation.

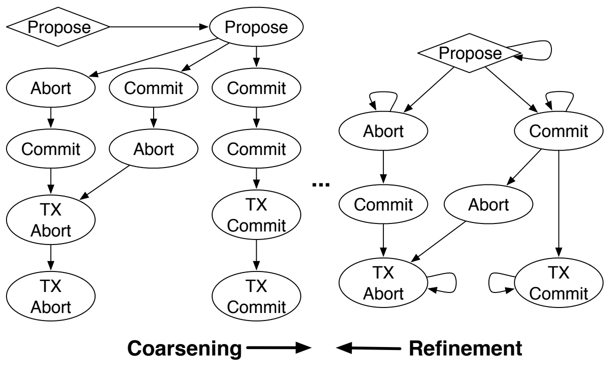

Figure 1: Two example representations of a two-node two-phase commit log. Arrows represent temporal ordering. Rhombus nodes are start states — that is, initial events. Coarsening can convert the left graph into the right one, which is smaller but admits more behaviors. Refinement is the dual of coarsening. Coarsening is the traditional approach to log summarization; this paper shows the benefits of refinement.

1 Introduction

Debugging distributed systems remains a significant challenge for developers who typically try to make sense of their systems by analyzing execution logs. However, even a small system with a few nodes running a distributed protocol, such as two-phase commit [15] or Paxos [22], can generate thousands of messages in just a few minutes of execution. Manual inspection of logs scales poorly, and a developer may easily miss an important system behavior. This paper presents *Synoptic* — a tool that summarizes system execution logs to support developer understanding of system behavior.

We approach the problem of understanding a system from its logs as a summarization and data-reduction challenge. Synoptic treats an input log as a set of event instances that can be grouped into event classes. Events may be related by a user-defined relation, such as time. Synoptic's goal is to produce a minimal relation graph with nodes representing sets of events and edges connecting nodes for which the user-defined relation holds. Figure 1 shows two Synoptic representations for a log containing two-phase commit protocol messages. This figure also illustrates two strategies for finding an appropriate summary log representation — refinement and coarsening. Synoptic mines a set of invariants in the form of LTL expressions from the log (e.g., Abort AlwaysFollowedBy TX Abort, in Figure 1), and uses model checking to enforce these invariants while exploring the representation space. This results in representations that retain the rich distributed system semantics present in the log.

Some prior work uses an FSM model to represent events as transitions and a coarsening procedure to derive a compact model. Synoptic differs in three ways, the latter two of which are novel to our work. (1) Synoptic uses a relational model to represent events as nodes in a graph. This better fits the log analysis domain. (2) Synoptic mines temporal invariants from the log to formulate intuitive event (dis-)similarity metrics and stopping conditions. This produces a smaller model that captures behavior better. (3) Synoptic utilizes both refinement and coarsening on the model. This improves efficiency by avoiding repeated operations on large data structures.

We applied Synoptic to real and synthetic logs of distributed systems. Compared to kTail [2], a popular coarsening algorithm, Synoptic produced smaller summarizations while preserving more key properties of the event log. Compared to a kTail version modified to preserve invariants, Synoptic has substantially better runtime performance. In a user study, Synoptic's output matched the mental model of a distributed systems developer and reminded him of specific corner-case design choices.

2 Approaching the Representations Space

Synoptic addresses the problem of finding a compact representation that summarizes a sequence of events logged by a distributed system. The notion of "event" depends on the system — events may be sent and received messages, local procedure invocations, debug output, or a combination of all of these. To run Synoptic, a user must (1) specify one or more partial orderings on the events (e.g., time) and (2) provide a way for Synoptic to identify events of the same class (e.g., message name).

Conceptually, the summarization problem can be broken down into two sub-problems — what type of representation model to use (Section 2.1) and how to explore the space of potential representations (Section 2.2).

2.1 Choice of Model

Synoptic uses a relational model. This section briefly overviews the advantages and disadvantages of state-based models, Petri nets, and the relational model. Space prevents us from discussing other potential models, such as HMMs [4].

State-based model. Distributed systems developers often structure node logic as a finite state machine (FSM) in which nodes represent system states [31, 5]. Although the FSM model is widely used, it is overly complex in our context because it forces an exploration algorithm to reason about system state. Events that appear in the log may indicate that the system is in a particular state, but we cannot assume that the log contains explicit state information. Reasoning about states detracts from the ultimate purpose of finding a representation that compactly summarizes the log.

Petri nets. Petri nets [28] provide a formal means to model and reason about concurrent systems. Their main advantage over FSMs is their explicit representation of concurrency and associated concepts like mutual exclusion. However, Petri nets are more difficult to understand and generate than FSMs and, in our experiments, the ability to express explicit concurrency did not improve the summarizations.

Relational model. The relational model captures possibly multiple relations between log events. For example, events may be related temporally, with the happened-before relation [21], or physically, by co-occurring at the same node. This model can be visualized as a graph in which each vertex represents a set of log events. A directed edge between two vertices indicates that the contained log events are related. Based on user feedback in a study we carried out (see Section 5.3) we have found this model to be the most appropriate for capturing log summaries. In addition, this model resembles a modal transition system, which is a natural fit for reasoning about temporal invariants between events. Finally, this model can represent multiple relations simultaneously (e.g., graphically by using edges of different colors/labels) and, unlike the state-based model, it makes minimal assumptions about the underlying process that produced the log events. In the rest of the paper, unless otherwise specified, we will assume a relational model.

The choice of model constrains how the exploration algorithm proceeds. It does not, however, severely limit the final representation shown to the user. For instance, we successfully experimented with mappings between the relational and state-based models, that make it possible to automatically convert representations between subsets of the two model types.

2.2 Exploring the Space

Now that we have decided to use the relational model, we must decide how to explore the large space of potential representations. We represent the relational model as a graph (V,E), such that V contains partitions, i.e., sets of events, and $E \subseteq V \times V \times R$ is a set of directed edges with labels from R, the set of available event relations. In the following, we use the term graph to mean this graph representation of the model.

Prior work in automata inference has used graph coarsening, which iteratively compresses a representation of the concrete trace by merging states in the FSM [2]. An alternate strategy is to perform graph refinement from an initial representation that consists of a single state. We now explain these two exploration directions in more detail.

2.2.1 Direction of Exploration

The space of potential representations can be explored using two dual operations: refinement and coarsening. These are illustrated in Figure 1.

Graph Coarsening. A step of graph coarsening produces a graph with fewer partitions than the previous graph. This new graph is produced by identifying at least two partitions containing similar information, and merging them. There exist a variety of partition similarity notions, such as behavioral and structural similarity.

Graph Refinement. A step of graph refinement produces a graph with at least one more partition than the prior graph. This new graph is produced by identifying a partition with events that are dis-similar enough to warrant separation of the events into different partitions.

Direction monotonicity. We call an algorithm that explore the space of representation in a single direction *monotonic*. The kTail algorithm [2] is a popular example of a monotonic *coarsening* algorithm. It typically starts from a graph in which each event (not event class) is mapped to its own partition, which embodies the starting assumption that all log events are different. In the simplest case, a monotonic *refinement* algorithm could start from a single-node graph, which embodies the starting assumption that all events are the same.

The rest of this paper focuses on two summarization algorithms. The first is *Bisim*, a monotonic refinement algorithm. The second is *BisimH*, a hybrid, non-monotonic algorithm that combines Bisim with kTail.

2.2.2 Guiding the Exploration with Policies

Two *policies*, termination and operation selection, induce a wide variety of exploration algorithms.

Termination Policy. An exploration algorithm must specify when a representation is considered final and no more coarsening/refinement exploration is performed.

Operation Selection Policy. Operation selection has two dimensions: (1) identifying candidate operations and (2) selecting a candidate as the next step. An operation either merges partitions or splits an existing partition.

To identify candidate operations, a summarization algorithm must determine whether two events are similar enough to warrant their representation as a single node and, when necessary, issue split/merge operations. This process crucially depends on the notion of similarity. If similarity is too fine-grained, the resulting representation will not be concise. If it is too coarse, the resulting graph might admit many spurious traces. If it is not an equivalence relation, the merge operation becomes non-associative, and the result not only depends on the operations made, but also on their order. For example, GK-Tail as specified in [24] is nondeterministic, because the output depends on the (unspecified) order of the operations.

Among all candidate operations, exactly one must be chosen for immediate processing. For example, consider the graph at E4 in Figure 8 (on the last page). Three splits are possible: two (edge 1,4 and edge 3,5) split partition A and one (edge 2) splits partition C. Thus, a graph exploration algorithm must define a policy that dictates which operation should be performed next.

We can express kTail [2] in our relational model in terms of the above two policies. kTail is a coarsening algorithm that starts with the most fine-grained representation. The *termination policy* is to stop once there is no pair of k-equivalent partitions, i.e., no two partitions that are roots of sub-graphs identical up to depth k. The *operation selection policy* merges any pair of k-equivalent partitions, chosen nondeterministically.

3 Bisim and BisimH Algorithms

Synoptic can use two algorithms to summarize a log. We first introduce the Bisim algorithm, a monotonic refinement algorithm. Then we present a key deficiency in Bisim to motivate BisimH, which is a hybrid algorithm combining Bisim and kTail to explore the space of representations non-monotonically.

```
1  Input: event log $L$
2  let initialGraph = extract($L$)
3  let $I$ = mineInvariants(initialGraph)
4  let $(V,E)$ = partition(initialGraph)
5  while $(V,E)$ does not satisfy invariants $I$
6     // p: event $\rightarrow$ boolean, $\pi$: partition that will be split
7     let $(p,\pi)$ = selectSplit($(V,E)$, $I$)
8     let $\pi_1 = \{\text{event} \in \pi \mid p(\text{event})\}$
9     let $\pi_2 = \{\text{event} \in \pi \mid \neg p(\text{event})\}$
10    $V := (V - \{\pi\}) \cup \{\pi_1, \pi_2\}$
11    $E := \{(\pi_3, \pi_4, r) \in V \times V \times R \mid \exists\, \text{event}_1 \in \pi_3, \exists\, \text{event}_2 \in \pi_4$
12          $:\ \text{event}_1\, r\, \text{event}_2 \in \text{initialGraph}\}$
13 end while
14 if (hybrid)
15    $(V,E)$ := kTail($(V,E)$, 0, $I$)
16 Output: $(V,E)$
```

Figure 2: The Bisim/BisimH algorithm, depending on the value of `hybrid`. The procedures `extract`, `mineInvariants`, and `partition` are described in Section 3.1.

3.1 The Bisim Algorithm

We developed *Bisim*, a bisimulation-inspired algorithm that combines partition refinement [9, 26] with model checking. Bisim is novel in finding a compact representation that satisfies a set of temporal relations from the event log. Figure 2 shows the Bisim pseudo-code, which involves four steps.

1. `extract()`, on line 2, creates an *initial graph* (step (b) in Figure 8) from the execution log. This graph contains a singleton partition for each event instance, and a directed edge between partitions that satisfy the partial ordering relation provided by the user.

2. `mineInvariants()`, on line 3, mines event relations listed in Figure 3 from the initial graph. We term these relations "invariants" because they succinctly capture temporal event relationships. Synoptic's mining approach is similar to that of Daikon [10]: it first enumerates all possible invariants, and then discards those invariants that are not satisfied by the initial graph. Due to space constraints, this paper omits optimization details that make this procedure efficient in practice.

3. `partition()`, on line 4, generates a *partition graph* (shown at E4 in Figure 8) from the initial graph by grouping events of the same class into a single partition and adding edges through existential abstraction. An edge between two partitions in this graph indicates that there are two events, one in each partition, that were connected in the initial graph. The partition graph may admit new traces that were not observed; however, each trace in the initial graph appears in the partition graph.

4. Bisim iteratively refines the partition graph — lines 5–13. This is necessary because newly-introduced traces may violate invariants. Bisim terminates when the graph satisfies all the mined invariants.

Invariant	LTL formula	Type
x AlwaysFollowedBy y	$\Box(x \rightarrow \Diamond y)$	liveness
y AlwaysPrecededBy x	$\Diamond y \rightarrow \neg y \ \mathrm{U}\ x$	safety
x NeverFollowedBy y	$\Box(x \rightarrow \Box\neg y)$	safety

Figure 3: Event relations that Synoptic mines from the input log, with corresponding LTL formula and classification. LTL properties must hold over the entire input log and are specified using the operators: *always* ($\Box$), *eventually* ($\Diamond$), and *until* (U). For example, the formula $\Diamond y \rightarrow \neg y \ \mathrm{U}\ x$ requires x to occur before y. Without the premise $\Diamond y$, x would be required to appear at least once, even if the trace does not contain y.

3.2 Bisim Policies

The Bisim algorithm is parametrized (Section 2.2.2) with respect to termination and split policies. Line 5 of Figure 2 fixes the termination criterion and line 7 uses `selectSplit` for split selection. This section motivates our choices of these policies and their effects.

3.2.1 Termination

Bisim uses a set of mined invariants for its termination criterion. Bisim's output is highly sensitive to this set. On the one hand, suppose the set of invariants is empty. Then the output is the quotient under label-equivalence, i.e., the initial partitioning. This graph is often too compact to capture key properties of the log. For example, when compacting correct two-phase commit event logs, the resulting graph incorrectly permits TX Commit to follow an Abort message. On the other hand, suppose that the invariant set includes all possible temporal log invariants expressible in LTL. Then the algorithm will terminate when, for all partitions A, if an event in A has a successor event in a partition B in the log, then every event in A has a successor event in B in the graph. In this case, the final representation is the quotient under bisimulation, i.e., a graph that satisfies the same set of LTL formulas as the log. Our experiments indicated that the bisimulation quotient is usually too similar to the initial graph, and thus too fine-grained to be a useful summary.

Bisim chooses a compromise between these extremes. It terminates when the graph satisfies the three types of mined invariants listed in Figure 3. These invariant types are partially based on the specification patterns formulated by Dwyer et al. [8] and are key to capturing rich semantics of distributed systems. For instance, key properties of two-phase commit and the Peterson algorithm (Section 5.1) are captured by these invariant types.

3.2.2 Operation Selection

Our overall goal for Bisim is to pick a sequence of splits such that the resulting graph is the coarsest graph that satisfies a set of invariants. This problem is NP-hard [6], so an efficient algorithm might not yield the optimal result. We describe Bisim's split selection policy in terms of the dimensions of exploration (Section 2.2.2) and explain how it may make suboptimal decisions.

Candidate Operations. Bisim converts the mined invariants expressed in LTL into Büchi automata using LTL2Büchi [13]. It then uses a model checker to obtain a set of counterexample traces for the current graph, each of which fails to satisfy at least one of the mined invariants. Next, Bisim follows the CEGAR approach [6] to determine a set of *candidate partitions*, for each of which there exists a split that removes at least one of the counterexamples. Bisim identifies these partitions heuristically by tracing each counterexample, stepwise, in parallel, in the initial graph and in the current graph. In the initial graph, only a prefix of the counterexample will be present (otherwise the counterexample would not violate an invariant). Bisim finds the longest such prefix, and the last partition of this prefix in the current graph is the candidate partition – it allows a spurious transition that makes the trace a counterexample.

To remove the counterexample, the events in a candidate partition are divided into two sets: a set that contains events that can perform the spurious transition in the initial graph and a set that cannot. In line 7 of Figure 2, `selectSplit` obtains a predicate p that distinguishes these two event sets, and lines 8 and 9 introduce two new partitions, π_1 and π_2, corresponding to these two sets.

In Synoptic's current implementation, `getSplit` only returns predicates that indicate the presence of an outgoing transition to a certain partition. Often, this constraint prevents Bisim from achieving an optimal splitting. To see this, consider Figure 8, in which Bisim yields a graph with six nodes at B6, while an invariant-satisfying graph with five nodes exists at C6. From the partition graph at E4, Bisim can obtain at most three different refinements, depending on which outgoing edge is used by the split. However, none of these refinements separates the event $\{a3\}$ from events $\{a1, a2, a4\}$, which is necessary to yield the optimal graph.

In the future, we plan to consider different choices of predicate p, such as ones that indicate the presence of incoming edges.

Ranking among Candidate Partitions. Typically, the refined graph violates several invariants and candidate partitions must be ranked to decide which one to split first. Currently, Synoptic employs a two-class ranking: it examines all counterexamples in arbitrary order and performs the first split that validates an invariant (i.e., eliminates the last counterexample for that invariant). If no such split is available (because there are several violating traces for each invariant), Bisim picks a split nondeterministically. This two-class ranking introduces nondeterminism and Bisim might perform unnecessary splits.

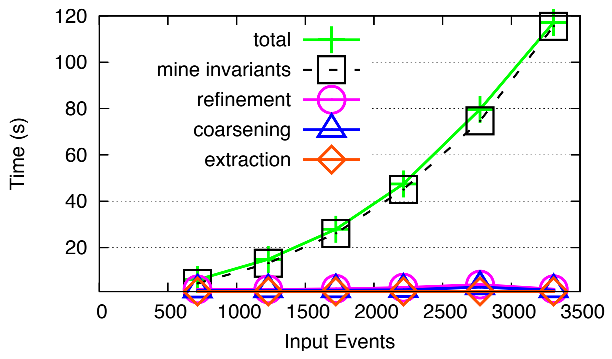

Figure 4: BisimH execution time on Peterson algorithm logs (Section 5.1) with varying number of events. The total execution time is broken up according to the steps described in Section 3.1.

3.3 The BisimH Algorithm

We have explained that the Bisim algorithm often refines more than it needs to. In particular, the result might contain partitions that can be merged without violating invariants. This seems to be inherent to the approach due to the NP-hardness of finding the optimal refinement [6].

Motivated by the counterexample in Figure 8, we developed *BisimH*. BisimH extends *Bisim* with one additional step that mitigates the unnecessary splits by merging all partitions that can be merged without violating invariants. BisimH runs *kTail* (described at the end of Section 2.2.2), with $k = 0$, on the refined graph while enforcing that all merges respect the invariants. The resulting *merged graph* is locally minimal: merging any two partitions will violate some invariant.

In the future, we plan to explore more complex hybrid strategies, such as interleaving coarsening and refinement phases or considering both coarsening and refinement operations at every step.

4 Performance Evaluation

Figure 4 shows the execution time of BisimH benchmarked on an AMD64 Intel i7 (1.6 GHz) Kubuntu machine with 8GB RAM. Each data point is the average of ten executions.

Performance vs. kTail with Invariants. kTail *without* invariants is usually much faster than BisimH but fails to produce compact graphs that satisfy key event invariants (see Figure 7). It is straightforward to modify kTail to check for invariant violations and terminate once further merges violate invariants. However, BisimH scales better than kTail with Invariants due to its use of refinement. For example, when run on a Peterson log with 716 events, kTail with Invariants takes 86 seconds while BisimH completes in less than 1 second. For kTail, model checking dominates the performance cost, while BisimH operates on much smaller graphs. In the Peterson example, BisimH begins with a graph of size 5, and issues only 35–54 splits during operation. Thus, the graphs BisimH must model check contain at most 60 nodes. This provides tremendous speed-up compared to kTail, which initially operates on graphs with 716 nodes.

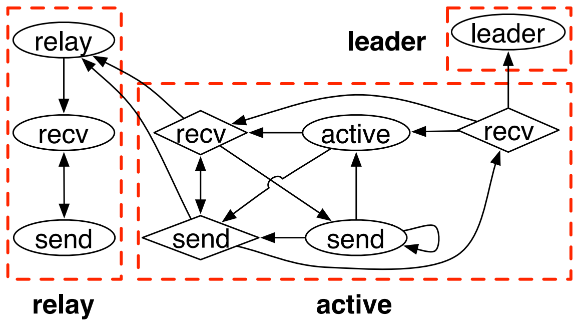

Figure 5: BisimH output for a Peterson log with 3308 events, generated by simulating 5 nodes. The manually-added, labeled, dotted regions group nodes into the states a node may take on in the algorithm.

5 Case Studies

To empirically evaluate Synoptic's ability to produce concise and useful representations, we applied it to three systems. The first system simulates the Peterson leader election algorithm [27] and produces a log of all messages exchanged between simulated nodes in the system. The second system is a Twitter client [34] that logs all messages to and from Twitter. The third system determines the likely reverse traceroute from an arbitrary destination on the Internet to a source host [20]. Its logs are debugging events generated by the coordinating server.

5.1 Peterson Leader Election

The Peterson leader election algorithm [27] allows an asynchronous unidirectional ring network of nodes to elect a leader. All nodes start as *active* and with a random unique node ID. In each round, at least half of the active nodes become *relays* through exchange and comparison of node IDs. A relay node forwards all messages it receives. When an active node receives its own messages via a ring of relay nodes, it becomes the *leader*.

We implemented a simulator of the Peterson algorithm that logs all messages sent and received by a node, as well as node state transition debug messages. This log includes a variety of message interleavings as the simulator allows concurrent node execution. Messages are timestamped and partially ordered using a Lamport vector clock [21]. Figure 5 shows BisimH output for an execution with 5 nodes that generated 3308 log events. This graph is useful in understanding node behavior as nodes take on different states. For example, Bisim correctly captures the fact that a relay node cannot send before receiving while an active node may first send and then receive, depending on the timing of incoming messages.

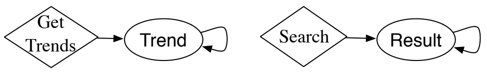

Figure 6: BisimH output for Twitter client-server interactions, when requesting trends (left) and performing a search (right).

5.2 Twitter API

We applied Synoptic to an open-source Twitter client [35]. We added 3 lines of code to this client to log all messages exchanged between the client and the Twitter server. We verified Synoptic's output by consulting the official Twitter API documentation [34]. Figure 6 shows two of the graphs from the final representation that demonstrate that Synoptic may help developers gain an intuitive understanding of the overall structure of an API.

5.3 Reverse Traceroute

We carried out a user study with a developer who built a system to determine the likely reverse traceroute from an arbitrary destination on the Internet to a source host [20]. Reverse traceroute relies on a distributed set of Internet vantage points and uses a variety of methods to find each segment of the reverse route, such as IP record route and timestamp options [17, 18], and relying on IP spoofing from PlanetLab hosts.

We developed a small parser that uses regular expressions to extract information from logs generated by Reverse Traceroute. We then partitioned the output of this parser into traces, such that each dealt with determining one segment of the reverse path. Synoptic executed on these partitioned logs. We then discussed the Synoptic-generated representation (which we omit due to lack of space) with the developer.

The Synoptic representation prompted us to ask relevant questions about the system's design and helped the developer to remember implementation details. The developer pointed out distinct regions of the graph where the system employed a particular method to determine a segment on the reverse path. The user stated that Synoptic makes it easy to discover unexpected paths in the system and that he found it useful to see how often certain paths were taken by the system (edges between events in the graph were annotated to indicate co-event frequency).

5.4 Graph Quality: BisimH vs. kTail

Figure 7 compares the graphs generated by BisimH, kTail, and kTail with Invariants (from Section 4).

Unlike kTail, BisimH was always able to establish the key invariants — those that imply system correctness. We do not know the key invariants for reverse traceroute, which, unlike the others, lacks a proof of correctness and documentation. Further, with kTail, the user must guess the value of the k parameter, which trades off the size and accuracy of the graph. BisimH makes explicit which invariants it preserves. Figure 7 also demonstrates that invariant types mined by BisimH (Figure 3) are sufficient to capture key invariants of a range of distributed systems.

Augmenting kTail to respect key invariants (kTail with Invariants in Figure 7) produces graphs that are as concise and as accurate as BisimH. However, BisimH is dramatically faster.

6 Discussion and Future Work

BisimH uses three types of invariants that we believe are useful for capturing temporal system properties. Two of these appear in the survey of specification patters [8]. Synoptic can accommodate other types of invariants that relate events. We plan to expand Synoptic to use the remaining six specification patterns from this survey. We also plan to expand Synoptic to mine structural properties of logged events, e.g., the value of the source and destination IP addresses and message data payload. Our initial experiments indicate that this direction is promising, though it may require users to specify format and data types of event fields.

BisimH is motivated by the Bisim counterexample in Figure 8. We are actively exploring the limits of BisimH and, thus far, have been unable to find a similar counterexample. However, one likely exists since finding an efficient, exact algorithm is equivalent to proving $P = NP$ [6]. Currently BisimH leverages kTail only once Bisim terminates, but another approach could interleave kTail and Bisim.

Synoptic's output depends on the ordering of events in the log. If events are missing, are out of order, or the log contains spurious events, Synoptic may mine false invariants or may omit true invariants, compromising the quality of the resulting graph. In small graphs, discrepancies could be easy to detect for users familiar with the system. In large graphs, manual detection is difficult. One way to handle log defects is to leverage anomaly detection to alert the user when an event occurrence or omission in the log may be due to a defect.

For Synoptic to converge on an accurate representation, it is sufficient for the log to contain an example trace corresponding to every path in the system model. For protocols and system that are well specified (e.g., two-phase commit) this is straightforward. For more complex systems, such as reverse traceroute, it is non-trivial to determine how much input is necessary. The intended use of Synoptic-generated models plays a role in determining the appropriate input log size. A more rigorous understanding of the system will require a larger log.

	Two-Phase Commit			**Peterson**			**Twitter**			**Reverse Traceroute**		
	24 events			716 events			87 events			2044 events		
	nodes	invs	time (ms)	nodes	invs	time (ms)	nodes	invs	time (ms)	nodes	invs	time (s)
kTail, k = 1	2	no	6	5	no	261	19	yes	8	21	no*	1.962
kTail, k = 2	7	yes	10	14	no	940	22	yes	18	37	no*	4.288
kTail with Inv.	7	yes	155	12	yes	3743s	19	yes	48269	-	-	>20000
BisimH	7	yes	110	9	yes	9310	19	yes	2414	114	yes*	13558

Figure 7: Graph quality metrics — number of input log events, output nodes, and whether the output satisfied the algorithm's key invariants (invs) — and execution times on logs of four systems. A * indicates that the key invariants were unknown, but, for reverse traceroute, kTail failed to preserve 68 invariants that BisimH mined and enforced.

7 Related Work

Work related to Synoptic falls into three main categories; (1) tools to debug and visualize distributed systems; (2) algorithms to create concise FSM representations of system executions; and (3) the study of bisimulations, which motivated our development of Bisim..

Debugging and Visualizing Distributed Systems. Distributed systems are notoriously difficult to get right. This is exemplified by recent efforts that target bug finding in distributed systems [38, 37]. Magpie [7] and X-Trace [12] are examples of debugging tools that provide a fine granularity of process tracing in distributed systems. Such tools, however, require system modification and do not focus on the problem of log summarization.

Prior work on distributed system log mining focused on detecting dependencies [25], anomalies [36, 19, 41], and performance debugging [32, 33]. Much of this work does not target the problem of finding a concise summary representation for an arbitrary distributed system. For instance, SALSA [32] and Mochi [33] extract and visualize node behavior of Hadoop [16] node logs to support performance debugging. This line of work is MapReduce-specific. Perracotta [40] mines and visualizes temporal properties of event traces, and it has been used to study program evolution [39]. Unlike Synoptic, Perracotta considers a totally ordered trace of events and does not consider properties that might be of interest in distributed systems, such as the invariants mined by Synoptic.

kTail and GK-Tail Algorithms. The problem of automata inference from positive examples of executions is computable [3], but is NP-complete [14, 1], and the FSA cannot be approximated by any polynomial-time algorithm [29]. Therefore, polynomial-time algorithms that explore the FSM space are approximation algorithms.

An important algorithm that has been used extensively in related work is the kTail algorithm [2]. kTail takes an automaton and produces a more compact one by recursively merging automaton states whose root subgraphs are identical up to a depth of k. Lorenzoli et al. [24] developed a variant of kTail, called GK-Tail, and applied it to logged sequences of method call invocations. Unlike Bisim and BisimH, the GK-Tail algorithm does not preserve trace invariants. Another use of kTail is in work by Lo et al. [23], in which temporal properties are mined from execution traces and are used to steer the kTail algorithm to ensure that a kTail merge will not produce a graph violating a temporal constraint. This is similar to the kTail with Invariants algorithm of Section 4.

This paper considers refinement exploration algorithms in contrast to the coarsening strategy in kTail.

Bisimulation. A bisimulation is a simulation relation that provides a strong notion of similarity for relational structures [30]. Its key property is preserving certain properties of the relational structure, for example, two strongly bisimilar transition systems are guaranteed to satisfy the same set of LTL formulas. An important application in model checking is model minimization [11]. Our Bisim algorithm is a modification of a partition refinement algorithm [26], which uses invariants to determine which state to split next and when to stop splitting, resulting in a coarser representation that is not bisimilar to the input structure. Our Bisim algorithm is also related to the partition refinement algorithms in [9], but Bisim uses invariants to guide exploration and termination.

8 Conclusion

Synoptic summarizes distributed system execution logs with a compact graph by employing two key innovations. First, Synoptic mines temporal properties from logged events and maintains these invariants during summarization. Compared to other approaches, such as kTail, Synoptic generalizes the logs more accurately and captures more key properties. Second, Synoptic uses coarsening to explore the representations space starting with smaller representations, making it scale better than the popular kTail algorithm augmented with invariant checking. A user study showed that Synoptic representations can augment the intuition of distributed systems developers. Synoptic is an open-source tool: `http://code.google.com/p/synoptic/`

Acknowledgments Josh Goodwin contributed to an early version of Synoptic. Financial support was provided by a Fulbright fellowship, NSF Grant #0937060 to the Computing Research Association for the CIFellows Project, and an IBM John Backus Award.

References

[1] ANGLUIN, D. Finding patterns common to a set of strings. *Journal of Computer and System Sciences 21*, 1 (1980), 46 – 62.

[2] BIERMANN, A. W., AND FELDMAN, J. A. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput. 21*, 6 (1972), 592–597.

[3] BLUM, L., AND BLUM, M. Toward a mathematical theory of inductive inference. *Information and Control 28*, 2 (1975), 125 – 155.

[4] CAPPÉ, O., MOULINES, E., AND RYDEN, T. *Inference in Hidden Markov Models (Springer Series in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[5] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation* (Berkeley, CA, USA, 1999), USENIX Association, pp. 173–186.

[6] CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided abstraction refinement. In *Computer Aided Verification* (2000), Springer, pp. 154–169.

[7] DOMINGUE, J., AND DZBOR, M. Magpie: supporting browsing and navigation on the semantic web. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces* (New York, NY, USA, 2004), ACM, pp. 191–197.

[8] DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, 1999), ACM, pp. 411–420.

[9] ELOMAA, T. Partition-Refining Algorithms for Learning Finite State Automata. In *ISMIS '02: Proceedings of the 13th International Symposium on Foundations of Intelligent Systems* (London, UK, 2002), Springer-Verlag, pp. 232–243.

[10] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, 1999), ACM, pp. 213–224.

[11] FISLER, K., AND VARDI, M. Y. Bisimulation Minimization and Symbolic Model Checking. *Formal Methods in System Design 21*, 1 (2002), 39–78.

[12] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-Trace: A Pervasive Network Tracing Framework. In *NSDI* (2007).

[13] GIANNAKOPOULOU, D., AND LERDA, F. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems* (London, UK, 2002), Springer-Verlag, pp. 308–326.

[14] GOLD, E. M. Language identification in the limit. *Information and Control 10*, 5 (1967), 447–474.

[15] GRAY, J. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course* (London, UK, 1978), Springer-Verlag, pp. 393–481.

[16] Welcome to Apache Hadoop!, `http://hadoop.apache.org/`. Accessed January 13, 2010.

[17] IPv4 Specification, Record Route option. `http://www.ietf.org/rfc/rfc791.txt`. Pg. 20, 21. Accessed March 9, 2010.

[18] IPv4 Specification, Timestamp option. `http://www.ietf.org/rfc/rfc791.txt`. Pg. 22, 23. Accessed March 9, 2010.

[19] JIANG, G., CHEN, H., UNGUREANU, C., AND YOSHIHIRA, K. Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. *Autonomic Computing, International Conference on 0* (2005), 111–122.

[20] KATZ-BASSETT, E., MADHYASTHA, H. V., ADHIKARI, V. K., SCOTT, C., SHERRY, J., VAN WESEP, P., ANDERSON, T., AND KRISHNAMURTHY, A. Reverse Traceroute. In *NSDI'10: : Proceedings of the 7th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2010), USENIX Association.

[21] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[22] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[23] LO, D., MARIANI, L. E., AND PEZZÈ, M. Automatic steering of behavioral model inference. In *ESEC/FSE '09: Proceedings of the the 7th joint*

meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (New York, NY, USA, 2009), ACM, pp. 345–354.

[24] LORENZOLI, D., MARIANI, L., AND PEZZÈ, M. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ACM, pp. 501–510.

[25] LOU, J.-G., FU, Q., WANG, Y., , AND LI, J. Mining Dependency in Distributed Systems through unstructured log analysis. In *WASL'09: Proceedings of USENIX Workshop on Analysis of System Logs* (2009).

[26] PAIGE, R., AND TARJAN, R. E. Three partition refinement algorithms. *SIAM J. Comput. 16*, 6 (1987), 973–989.

[27] PETERSON, G. An O (n log n) unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 4 (1982), 762.

[28] PETERSON, J. L. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[29] PITT, L., AND WARMUTH, M. K. The minimum consistent DFA problem cannot be approximated within any polynomial. *J. ACM 40*, 1 (1993), 95–142.

[30] SANGIORGI, D. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst. 31*, 4 (2009), 1–41.

[31] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv. 22*, 4 (1990), 299–319.

[32] TAN, J., PAN, X., KAVULYA, S., G, R., AND NARASIMHAN, P. SALSA: Analyzing Logs as StAte Machines. In *WASL'08: Proceedings of USENIX Workshop on Analysis of System Logs* (2008).

[33] TAN, J., PAN, X., KAVULYA, S., G, R., AND NARASIMHAN, P. Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop. In *HotCloud'09: Proceedings of USENIX Workshop on Hot Topics in Cloud Computing* (2009).

[34] Twitter API Documentation, `http://apiwiki.twitter.com/Twitter-API-Documentation`. Accessed June 10, 2010.

[35] A Java wrapper around the Twitter API, `http://code.google.com/p/java-twitter/`. Accessed March 8, 2010.

[36] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 117–132.

[37] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: predicting and preventing inconsistencies in deployed distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2009), USENIX Association, pp. 229–244.

[38] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2009), USENIX Association, pp. 213–228.

[39] YANG, J., AND EVANS, D. Automatically Inferring Temporal Properties for Program Evolution. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 340–351.

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

[41] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: error diagnosis by connecting clues from run-time logs. *SIGARCH Comput. Archit. News 38*, 1 (2010), 143–154.

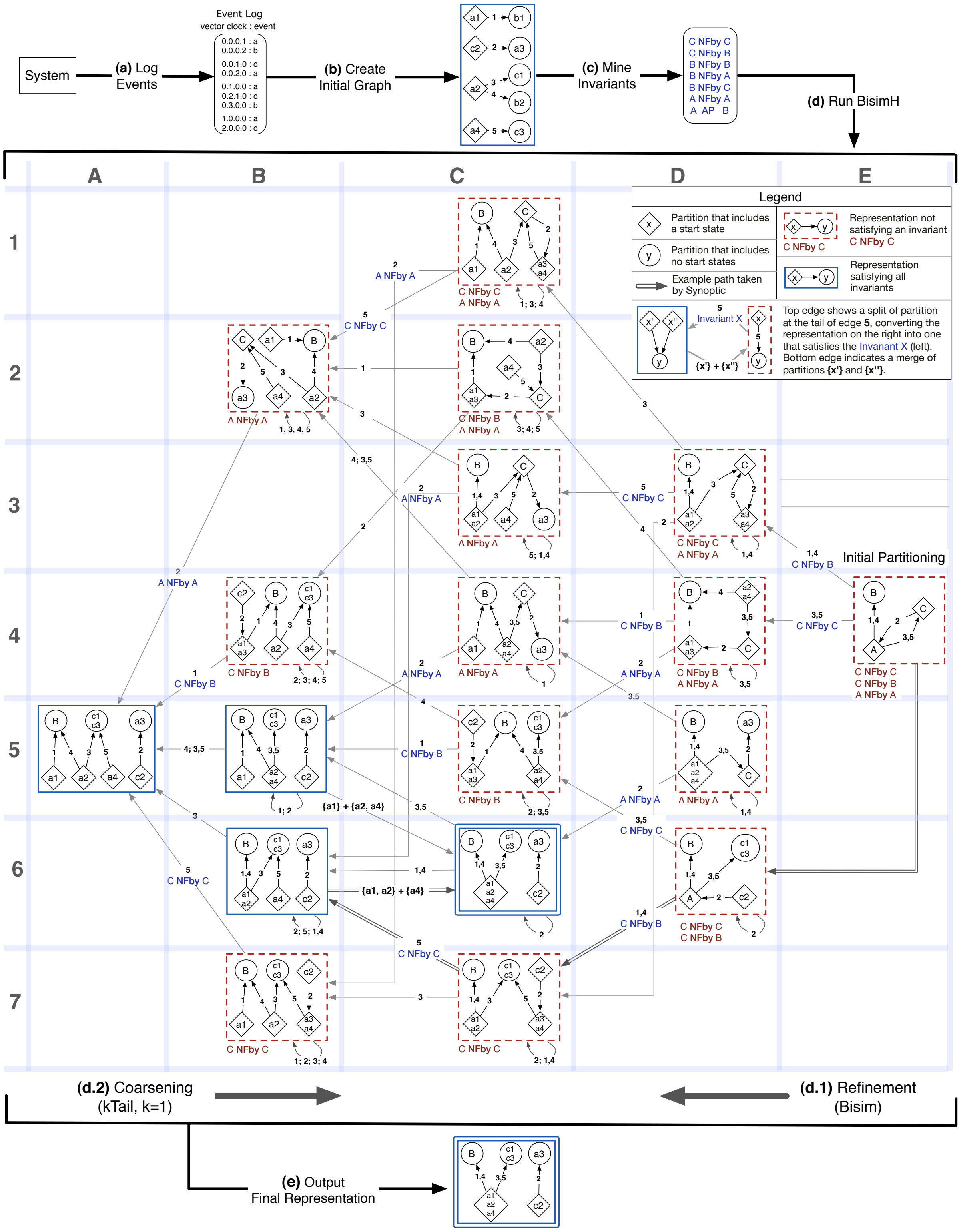
System
(a) Log Events
Event Log
vector clock : event
0.0.0.1 : a
0.0.0.2 : b
0.0.1.0 : c
0.0.2.0 : a
0.1.0.0 : a
0.2.1.0 : c
0.3.0.0 : b
1.0.0.0 : a
2.0.0.0 : c
(b) Create Initial Graph
(c) Mine Invariants
C NFby C
C NFby B
B NFby B
B NFby A
B NFby C
A NFby A
A AP B
(d) Run BisimH
A
B
C
D
E
1
2
3
4
5
6
7
Legend
Partition that includes a start state
Partition that includes no start states
Example path taken by Synoptic
Representation not satisfying an invariant
C NFby C
Representation satisfying all invariants
Invariant X
{x'} + {x''}
Top edge shows a split of partition at the tail of edge 5, converting the representation on the right into one that satisfies the Invariant X (left). Bottom edge indicates a merge of partitions {x'} and {x''}.
Initial Partitioning
{a1} + {a2, a4}
{a1, a2} + {a4}
(d.2) Coarsening (kTail, k=1)
(d.1) Refinement (Bisim)
(e) Output Final Representation

Figure 8: Step-by-step depiction of Synoptic's process: **(a)** log events; **(b)** derive the *initial graph*; **(c)** mine temporal relations (NFby: NeverFollowedBy, AFby: AlwaysFollowedBy, AP: AlwaysPrecedes); **(d)** explore the representation space with BisimH. BisimH has two phases: **(d.1)** refine (right to left) from the initial partitioning at E4 until a representation satisfying all the mined invariants at B6; **(d.2)** coarsen (left to right) towards the final, more compact, representation at C6, which retains the satisfied invariants. Finally, **(e)** the derived representation is presented to the user. Bisim is sub-optimal on this example, motivating BisimH. The most compact valid representation at C6 *cannot be reached by refinement alone* when starting at E4.