Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models

Ivan Beschastnikh[†] Yuriy Brun[†] Sigurd Schneider[ꝏ] Michael Sloan[†] Michael D. Ernst[†]
[†]Computer Science & Engineering, University of Washington
[ꝏ]Computer Science, Saarland University
{ivan,brun,mgsloan,mernst}@cs.washington.edu, sigurd@ps.uni-saarland.de

Abstract

Computer systems are often difficult to debug and understand. A common way of gaining insight into system behavior is to inspect execution logs and documentation. Unfortunately, manual inspection of logs is an arduous process and documentation is often incomplete and out of sync with the implementation.

This paper presents *Synoptic*, a tool that helps developers by inferring a concise and accurate system model. Unlike most related work, Synoptic does not require developer-written scenarios, specifications, negative execution examples, or other complex user input. Synoptic processes the logs most systems already produce and requires developers only to specify a set of regular expressions for parsing the logs.

Synoptic has two unique features. First, the model it produces satisfies three kinds of temporal invariants mined from the logs, improving accuracy over related approaches. Second, Synoptic uses refinement and coarsening to explore the space of models. This improves model efficiency and precision, compared to using just one approach.

In this paper, we formally prove that Synoptic always produces a model that satisfies exactly the temporal invariants mined from the log, and we argue that it does so efficiently. We empirically evaluate Synoptic through two user experience studies, one with a developer of a large, real-world system and another with 45 students in a distributed systems course. Developers used Synoptic-generated models to verify known bugs, diagnose new bugs, and increase their confidence in the correctness of their systems. None of the developers in our evaluation had a background in formal methods but were able to easily use Synoptic and detect implementation bugs in as little as a few minutes.

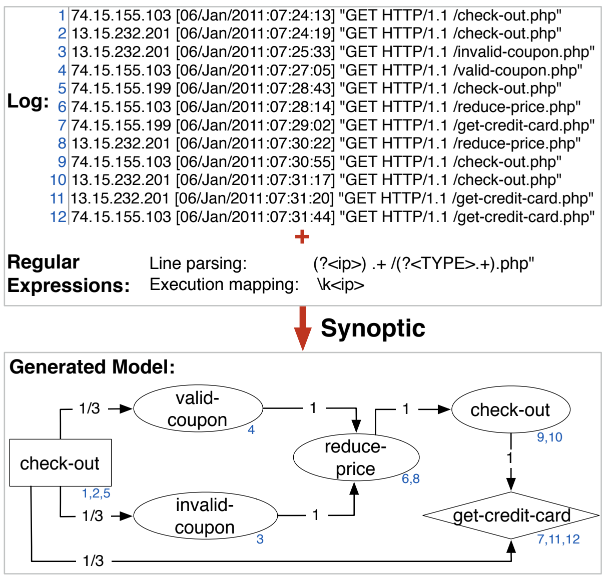
Categories and Subject Descriptors: D.2.5 [Testing and Debugging]: Debugging aids
General Terms: Algorithms, Reliability
Keywords: log analysis, temporal invariant mining, model inference, Synoptic

1. Introduction

Application of formal methods, such as specification and verification of systems during the design stage, is a promising means of preventing implementation bugs. Although such formal methods are popular in limited domains (e.g., avionics), only rarely do tools from the formal methods community get used by everyday developers. More commonly, developers rely on debugging.

Logging system behavior is one of the most ubiquitous, simple, and effective debugging tools. Developers instrument key locations in the code to gain insight into the state of a process, the execution sequence, and the presence or absence of certain events. Logging is so important that production systems at companies like Google are instrumented to generate *billions* of log events each day. These events are stored for weeks to help diagnose bugs [45].

However, logs are analyzed primarily by hand or with ad hoc tools. The goal of our work is to infer concise, accurate models of system behavior from logs. The inferred models aim to help developers in three ways: (1) help find bugs, (2) increase developers' confidence in the absence of certain bugs, and (3) improve developers' understanding of their systems. We present the design and evaluation of a tool called Synoptic, which processes the logs most systems


Log:
```
 1 74.15.155.103 [06/Jan/2011:07:24:13] "GET HTTP/1.1 /check-out.php"
 2 13.15.232.201 [06/Jan/2011:07:24:19] "GET HTTP/1.1 /check-out.php"
 3 13.15.232.201 [06/Jan/2011:07:25:33] "GET HTTP/1.1 /invalid-coupon.php"
 4 74.15.155.103 [06/Jan/2011:07:27:05] "GET HTTP/1.1 /valid-coupon.php"
 5 74.15.155.199 [06/Jan/2011:07:28:43] "GET HTTP/1.1 /check-out.php"
 6 74.15.155.103 [06/Jan/2011:07:28:14] "GET HTTP/1.1 /reduce-price.php"
 7 74.15.155.199 [06/Jan/2011:07:29:02] "GET HTTP/1.1 /get-credit-card.php"
 8 13.15.232.201 [06/Jan/2011:07:30:22] "GET HTTP/1.1 /reduce-price.php"
 9 74.15.155.103 [06/Jan/2011:07:30:55] "GET HTTP/1.1 /check-out.php"
10 13.15.232.201 [06/Jan/2011:07:31:17] "GET HTTP/1.1 /check-out.php"
11 13.15.232.201 [06/Jan/2011:07:31:20] "GET HTTP/1.1 /get-credit-card.php"
12 74.15.155.103 [06/Jan/2011:07:31:44] "GET HTTP/1.1 /get-credit-card.php"
```
+

Regular Expressions:
```
Line parsing:        (?<ip>) .+ /(?<TYPE>.+).php"
Execution mapping:   \k<ip>
```

Figure 1: (Top) A log with line numbers for an online shopping cart, and two complete regular expressions for processing this log with Synoptic. (Bottom) The generated Synoptic model. In the model, rectangular/diamond/oval nodes indicate initial/terminal/intermediate nodes. Edge labels indicate transition probabilities. The subscript to the right of each node lists the log line numbers corresponding to the partition. This application contains a bug that is easily noticed in the generated model: processing an invalid coupon incorrectly reduces the shopping cart's total price.

already produce and requires developers only to specify a set of regular expressions for parsing the logs. Synoptic generates a model similar to a finite state machine that satisfies the temporal invariants mined from the logs. By leveraging existing logs, Synoptic produces models with details the developer already considers to be helpful in aiding understanding. More generally, Synoptic offers developers who have little expertise in formal methods a means to consider their systems more formally. Synoptic bridges the gap between the culture of the average developer who practices logging for debugging with advanced techniques developed by the formal methods community.

Synoptic differs from prior model-generation tools by its versatility and by imposing few requirements on the developer. To use the tool, developers neither need to specify their systems as part of the design, identify properties for the tool to verify, nor modify their code. Instead, Synoptic mines three kinds of temporal invariants from existing logs and uses these to generate a concise model satisfying the invariants. Developers only provide a set of regular expressions to parse events from the logs. With this approach, Synoptic (1) does not restrict developers to a particular log format and (2) allows developers to specify the events to include in the model.

Figure 1 shows a web server log for a shopping cart application. Using the two listed regular expressions, Synoptic parses the log into three traces, one for each of the three user IP addresses accessing the server (Figure 3). Synoptic then mines temporal invariants that hold in those traces and uses the invariants to infer a model of the system (bottom of Figure 1). The model clearly illustrates a bug that would be difficult to find by examining the log directly: applying an invalid coupon allows the user to reduce the price. Not only can Synoptic help a developer find this bug, it can also increase the developer's confidence that the bug has been successfully removed. For example, the developer can run Synoptic on logs generated by a new version of the system and compare the new model with the prior model.

We evaluated Synoptic both theoretically and experimentally. Section 3 formally proves that Synoptic produces a model that satisfies all the true temporal invariants mined from the log and none of the invariants that are not satisfied by the log. Further, we argue that Synoptic's exploration of the model space is efficient and produces concise models.

Additionally, to demonstrate Synoptic's ability to produce useful representations in practice, we evaluated it in two user experience studies (Section 4). We first report on a study with a developer working on reverse traceroute [23], a distributed system that determines the likely reverse Internet route between two hosts. Reverse traceroute has been in deployment for over 7 months, has handled a total of 3.6 million requests to date, and has been recently internally deployed by a large, popular, and ubiquitous Internet company. Second, we report on the experiences of 45 undergraduate students who used Synoptic in a distributed systems course. The students applied Synoptic to logs generated by their implementations of a distributed version of a cache coherence protocol [26].

In our evaluation, Synoptic generated models on logs up to 900,000 events that represent over 28,000 unique system executions. Most developers in our studies found the generated models helpful in understanding their systems. Synoptic models increased developer confidence in the correctness of their implementations, helped identify previously unknown bugs, and confirmed the existence of known bugs.

Next, we motivate and explain how Synoptic works by describing BisimH, the central algorithm Synoptic uses.

2. BisimH: Generating models from logs

Synoptic uses a hybrid refinement and coarsening algorithm called BisimH. Our prior workshop paper proposed this algorithm without

```
1   Input: log $L$, regular expressions $RegExps$
2   let traceGraph = extract($L$, $RegExps$)
3   let $I$ = mineInvariants(traceGraph)
4   let $(V,E)$ = partition(traceGraph)
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              $: \text{event}_1\ r\, \text{event}_2 \in \text{traceGraph}\}$
13  end while
14  $(V,E)$ := kTail($(V,E)$, 0, $I$)
15  Output: $(V,E)$
```

Figure 2: The BisimH algorithm. Section 2 describes the `extract`, `mineInvariants`, `partition`, `selectSplit`, and `kTail` procedures.

providing a formal analysis or reporting on user experience with the tool [39]. The rest of this section explains the algorithm in detail by walking through its pseudo-code listed in Figure 2, and by illustrating how Synoptic would process the log in Figure 1. For a more formal treatment see Section 3.

2.1 Log parsing

Synoptic constructs a system model from a set of observed system execution traces. It takes as input a log file containing the execution traces, and a set of user-defined regular expressions. Synoptic uses the regular expressions to parse the log file and extract from some of the log lines an *event instance*: a triplet containing: (1) a trace identifier, (2) a timestamp, and (3) an *event type*. Trace identifiers are used to group together event instances from the same *trace*. Synoptic requires that the event instances in a trace be totally ordered using their timestamps. Therefore, no two event instances in a trace may have identical timestamps. An event type can be an arbitrary string, and is usually defined by the developer as something that conveys important information about the system. For example, Section 4 presents two Synoptic-generated models in which an event type represents (1) an executed method's name, and (2) the state of a node in a distributed system.

A trace can be considered to be a linear graph — each vertex is an event instance, and the edges represent the total ordering. We term the union of such graphs a *trace graph*. The trace graph is built from the log using the provided regular expressions (line 2 in Figure 2).

Recall the shopping cart application. Figure 1 shows the log and the two complete regular expressions that Synoptic uses to parse the log into three traces, one per unique IP in the log; the php script names denote the trace event types. Figure 3 shows the three traces parsed from the log. For example, the trace corresponding to the IP 74.15.155.103 is $\langle 0$, check-out$\rangle$, $\langle 1$, valid-coupon$\rangle$, $\langle 2$, reduce-price$\rangle$, $\langle 3$, check-out$\rangle$, $\langle 4$, get-credit-card$\rangle$. Here, the integer timestamp is derived implicitly from the order of lines in the log.

2.2 Mining invariants from the trace graph

To guide model generation, Synoptic mines three kinds of temporal invariants relating *event types* from the trace graph (line 3 in Figure 2):

- **a Always Followed by b** (written $a \rightarrow b$). Whenever the event type a appears, the event type b always appears later in the same trace.
- **a Never Followed by b** (written $a \not\rightarrow b$). Whenever the event

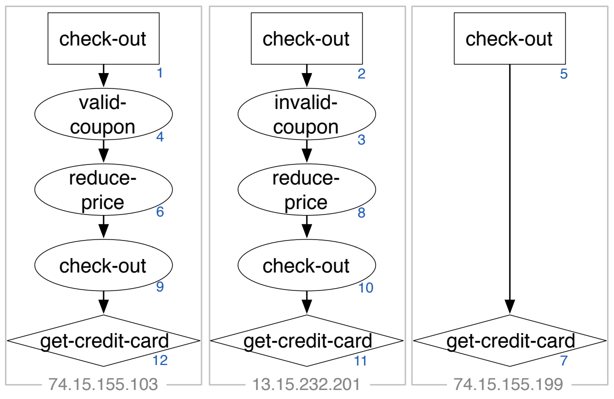


Figure 3: Trace graph parsed from the log in Figure 1. Each execution corresponds to an IP address that accessed the web application. The subscript to the right of each node lists the line number of the log line from which the event instance was extracted.

type a appears, the event type b never appears later in the same trace.

- **a Always Precedes b** (written $a \leftarrow b$). Whenever the event type b appears, the event type a always appears before b in the same trace.

The missing symmetrical invariant *Never Precedes*, defined as a Never Precedes b iff b can be generated only when no a was yet generated, is equivalent to the *Never Followed by* invariant.

We term these relations “invariants” because they succinctly capture temporal event type relationships that must hold true over all the input traces. The trace graph in Figure 3 yields 27 such invariants. Two examples are *reduce-price* $\not\rightarrow$ *valid-coupon*, and *invalid-coupon* $\rightarrow$ *check-out*. Section 2.5 justifies our use of these particular invariant types, and Section 3.2 explains how these invariants are mined. Next, we introduce Synoptic models.

2.3 Synoptic models and the initial model

The Synoptic model is a *partition graph* of the trace graph. Given a partitioning of the original vertices, each vertex in the model is one partition. Directed edges in the model are formed through existential abstraction. That is, a directed edge between two vertices indicates that there exists a pair of event instances in the corresponding partitions that are connected by an edge in the trace graph. A further constraint is that each partition contains event instances of only one particular event type. The resulting relational model makes minimal assumptions about the underlying process that produced the logged event instances. For a more complete discussion concerning our model choice see [39].

An important property of Synoptic models is that each trace in the input log is accepted by a model constructed from the corresponding event instances (in the sense that each trace maps to a valid path in the model). However, a Synoptic model is also generative — it may accept traces that were not present in the log.

The BisimH algorithm starts with an *initial model* (constructed using `partition` on line 4 of Figure 2). In this model, there is one partition per event type containing all the event instances of that type. Figure 4 shows the initial model for the trace graph in Figure 3.

By construction, the initial Synoptic model captures two important kinds of temporal properties for any two adjacent event instances in a trace in the log. First, if an event instance of type a is at some point immediately followed by an event instance of type b in the log, then there must be an edge from a to b. Second, if an event instance of type a is never immediately followed by an event instance of type b in the log, then there is no edge from a to b.

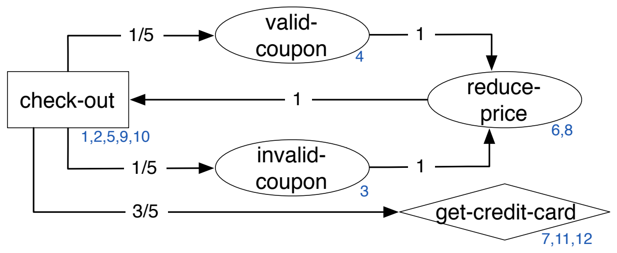


Figure 4: Initial model corresponding to the trace graph in Figure 3.

The initial model is therefore the most compact or abstract model plausible, based on the logged traces. The least compact (and most concrete) model is the trace graph, in which each partition contains a single event instance. This model makes no generalizations and overfits to the input traces.

2.4 Refinement and coarsening

Coarsening and refinement are dual operations on a Synoptic model. Starting with the initial model, Synoptic first performs model refinement, shown as an iterative process in lines 5–13 of Figure 2. This algorithm is a modification of a partition refinement algorithm introduced by Elomaa [13]. Synoptic refines (i.e., splits) partitions until it reaches a model that satisfies all the mined invariants. Next, Synoptic uses coarsening to merge those partitions that were needlessly refined due to an imperfect splitting heuristic (line 14 in Figure 2). The coarsening step is constrained to not violate the mined invariants satisfied during refinement. Synoptic outputs the model when it is unable to coarsen it any further.

2.4.1 *Refinement*

The refinement goal of BisimH is to pick a minimal sequence of splits, so that the resulting graph is the coarsest graph that satisfies a set of invariants. This problem is NP-hard [7], so an efficient algorithm might not yield the optimal result. For an example illustrating refinement suboptimality see [39].

BisimH performs splits as long as there exists some mined invariant that is not satisfied. BisimH uses an FSM-based model checker to check whether a model satisfies a mined invariant. It converts each invariant into a small FSM that accepts traces satisfying the invariant. It then updates the FSMs as it traverses the model graph. If the model does not satisfy an invariant, the model checker outputs a counterexample path. For example, the invariant *valid-coupon* $\not\rightarrow$ *invalid-coupon* mined from the log in Figure 1 is not true in the model in Figure 4 — Figure 5 shows a counterexample path.

Having identified a set of counterexamples that violate the mined invariants, BisimH follows the counterexample guided abstraction refinement (CEGAR) approach [7] to determine a set of *candidate partitions*, for each of which there exists a split that removes at least one of the counterexamples. BisimH identifies these partitions heuristically by tracing each counterexample, stepwise, in parallel, in the input traces and in the model. In the traces, only a prefix of the counterexample path will be present (otherwise the counterexample would not violate an invariant). BisimH finds the longest such prefix, and the last partition of this prefix in the model becomes a candidate for refinement — this partition allows a spurious transition in the model that allows for the counterexample path to exist. For example, the longest such prefix for the counterexample path in Figure 5 ends in the *check-out* partitions. This is because *check-out* stitches together two traces from the log (two left-most traces in Figure 3) into a trace that violates the *valid-coupon* $\not\rightarrow$ *invalid-coupon* invariant.

To refine a candidate partition (i.e., to eliminate the counterex-

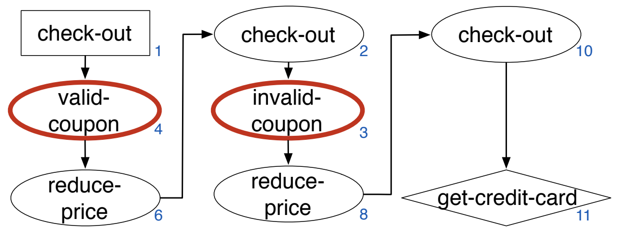


Figure 5: A path through the initial model in Figure 4 that violates the mined *valid-coupon* $\not\to$ *invalid-coupon* invariant.

ample path), the event instances in this partition are divided into two sets based on whether they can or cannot be reached from the partition immediately preceding the candidate partition in the prefix. In line 7 of Figure 2, `selectSplit` obtains a predicate p that distinguishes these two event instance sets, and lines 8 and 9 introduce two new partitions, π_1 and π_2, corresponding to these two sets. Figure 6 illustrates a refinement of the initial model in Figure 4 to eliminate the counterexample in Figure 5. In this case, the predicate p separates the event instances in the *check-out* partition into those that can or cannot be reached from the *reduce-price* partition.

We experimented with two kinds of predicates. Synoptic uses the one described above: it separates event instances in the candidate partition based on an incoming edge from a partition that immediately precedes the candidate partition. We also tried a predicate that separates event instances in the candidate partition based on an outgoing edge representing the spurious transition — it separates event instances in the candidate partition into sets based on whether they can or cannot make the spurious transition. Though we did not show this formally, in practice, we found the second strategy to be less optimal than the first. It is also possible to split the candidate partition simultaneously on an incoming and on an outgoing edge. Though we have not tried this, we think this may work best. In our future work, we intend to further study the splitting predicate's impact on the algorithm.

Typically, the refined model violates several invariants and candidate partitions must be ranked to decide which one to split first. Synoptic employs a two-class ranking: it examines all counterexamples in an 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 more counterexamples exist for each invariant), BisimH picks a split nondeterministically. This ranking introduces nondeterminism and BisimH might perform unnecessary splits.

2.4.2 *Coarsening*

BisimH may end up refining more than it needs to. When this happens, the model will contain partitions that can be merged without violating the satisfied invariants. After refinement, BisimH coarsens the model to merge such partitions (line 14 in Figure 2).

For coarsening, BisimH uses *kTail-equivalence* [5]. kTail is a coarsening algorithm that starts with the most fine-grained model. It stops 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. At each step, the algorithm merges one pair of *kTail-equivalent* partitions, chosen nondeterministically. BisimH runs kTail with $k = 0$ (label equivalence) to produce the most concise models. It starts with the final refined graph, under the extra constraint that all merges do not unsatisfy any invariants. The resulting merged model is locally minimal: merging any two partitions will violate some invariant.

2.5 The impact of mined invariants on BisimH

BisimH uses the mined invariants to establish a well-defined termination criterion for refinement, and also to guide refinement in its choice of partition to refine. This use of invariants is an important feature of BisimH. To see this, suppose the set of invariants is empty. In this case, refinement would terminate with a model that is the quotient under label-equivalence, i.e., the initial model. This model is often too compact to capture key properties of the log and is overly generative. 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 instance in A has a successor event instance in a partition B, then every event instance in A has a successor event instance in B in the model. In this case, the final model is the quotient under bisimulation, i.e., a graph that satisfies the same set of LTL formulae as the trace graph. In our experience, the bisimulation quotient is usually too similar to the trace graph, and thus too fine-grained to be considered concise.

Our choice of the three invariant types is a compromise between the above two extremes. In our experience, the models derived using this set of invariants are accurate, yet sufficiently generative for the kinds of applications we are considering (e.g., improving developer understanding of how the system operates). These invariant types are also exactly the most frequently observed specification patterns formulated by Dwyer et al. [12], with scope constrained to a trace (i.e., global scope). The translation is not one-to-one: $a \to b$ is Dwyer's Existence pattern when a is *START* (see Definition 2 below), and is otherwise Dwyer's Response pattern. Another example is $\forall b, a \leftarrow b$, which is Dwyer's Universality pattern. In our experience, these invariants were sufficient for capturing key temporal properties of the systems that produced the logs we considered.

Users can write Java code to define custom Synoptic invariants. However, all of the users in our case studies (Section 4) successfully used Synoptic without even knowing about its use of invariants.

In the next section, we define log and model formalism and prove important positive results about the BisimH algorithm.

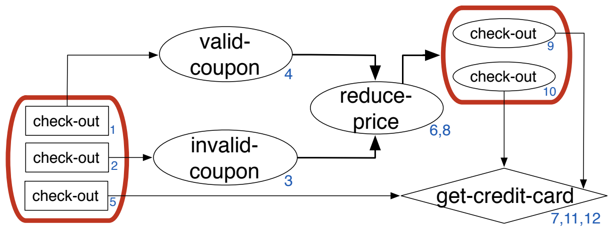


Figure 6: A refinement of the model in Figure 4 that eliminates the counterexample path in Figure 5. The edge between the *reduce-price* partition and the *check-out* partition induces a split of *check-out*: the *check-out* event instances reachable from *reduce-price* are split out. The two new *check-out* partitions, with the contained *check-out* event instances, are shown in bold. This model is equivalent to the final model shown at the bottom of Figure 1.

3. Formal evaluation

This section proves the correctness of our algorithm, and explains why it is efficient and is able to infer *concise* models in practice. Sections 3.1 and 3.2 define the formalisms. Section 3.3 proves that BisimH always halts and that the final model satisfies exactly the invariants mined from the input log. Section 3.4 proves an important result for improving model search efficiency, and Section 3.5 deals with model size.

3.1 Definitions

Two special event types — *START* and *END* — are added internally by Synoptic to keep track of initial and terminal events in the traces.[1]

Definition 1 (Event Types)**.** A set of event types is a finite set (alphabet) $E \supseteq \{START, END\}$.

Definition 2 (Trace)**.** Let E be a set of event types. Then for all $n \in \mathbb{N}_{\geq 2}$, a finite trace is an ordered sequence of event types $l \in E^n$ such that the first element of l is *START* and the last element is *END*. The length of l is $n-2$.

Definition 3 (Log)**.** A log L is a set of traces.

The set of event instances in a log is the collection of elements in the traces in that log. Each trace element is a unique event instance, indexed by its trace and position within that trace.

Definition 4 (Event Instances)**.** Let E be a set of event types. Let L be a log over E. Then an *event instance* is a triplet $\langle e,l,i\rangle$ such that $e \in E$ occurs in the log trace $l \in L$ at position $i \in \mathbb{N}$. $\hat{E}$ denotes the set of all such event instances for L.

An event instance relation is a set of pairs of elements of $\hat{E}$. For example, one representation (using "$\mathbf{0}$" to represent the event instance $\langle \mathbf{0}, l, 1\rangle$, etc.) of the event instance relation "next" on the log trace $l = \langle 0,1,2,3,4\rangle$ is $\{\langle 0,1\rangle, \langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle\}$. This paper's examples only use this "next" relation, although all the results generalize to arbitrary relations.

Definition 5 (Event Instance Relation)**.** Let $\hat{E}$ be a set of event instances. Then $r \subseteq \hat{E}^2$ is an event instance relation.

A partitioning of a finite set of event instances $\hat{E}$ is a finite set of disjoint, exhaustive subsets of $\hat{E}$. Each subset is called a partition and contains event instances of the same event type.

Definition 6 (Partitioning)**.** Let $\hat{E}$ be a set of event instances. Then $P \subset \mathcal{P}(\hat{E})$ is a partitioning of $\hat{E}$ if $\forall$ distinct $p,q \in P$, $p \cap q = \emptyset$, and $\hat{E} = \bigcup_{p \in P} p$, and $\forall p \in P$, all $\hat{e} \in p$ are of the same event type. Each $p \in P$ is called a partition. We enforce the condition that for a valid partitioning, all instances of the *START* event type are in a single partition and all instances of the *END* event type are in a single partition. That is, for all $\hat{e_1} \in p_1$, $\hat{e_2} \in p_2$: $\hat{e_1}, \hat{e_2}$ instances of *START* $\Rightarrow p_1 = p_2$ and $\hat{e_1}, \hat{e_2}$ instances of *END* $\Rightarrow p_1 = p_2$.

A relational model is a partition graph. The largest (most nodes) relational model for a log L is the trace graph: the set of disconnected subgraphs, one for each trace $l \in L$, with a vertex for each event instance and edges only between consecutive event instances in l. Other relational models can be generated by merging vertices that represent event instances of the same event type (and removing redundant edges).

Definition 7 (Relational Model)**.** Let $\hat{E}$ be the set of event instances in a log. Let R_r be a family of relations over $\hat{E}$ indexed by r. Then the relational model is a directed graph $M = \langle M_V, M_A\rangle$, such that M_V is a partitioning of $\hat{E}$ and $a = \langle p_1, p_2, r\rangle \in M_A \subseteq M_V \times M_V \times R_r$ iff $p_1, p_2 \in M_V$ and $\exists \hat{e}_1, \hat{e}_2 \in \hat{E}$ such that $\hat{e}_1 \in p_1$, $\hat{e}_2 \in p_2$, and $\langle \hat{e}_1, \hat{e}_2\rangle \in R_r$.

Definition 8 (Complete Path)**.** A path in a model is complete if it starts at the *START* partition and ends at the *END* partition.

A relational model M accepts a trace if the event instances of the trace form a complete path in M.

[1]When viewing a model, the user can optionally hide these nodes, and instead have Synoptic specially mark the partitions containing any initial and terminal events as rectangles and rhombuses, respectively. The models pictured in this paper were all generated in this way.

Definition 9 (Trace Acceptance)**.** Let $n \in \mathbb{N}$ and let $l = \langle START$, $\hat{e}_1, \ldots, \hat{e}_n$, $END\rangle$ be a trace of length n. Then a relational model M accepts l iff $\exists \Pi = \langle p_{START}, p_1, \ldots, p_n, p_{END}\rangle$ such that $\forall 1 \leq i \leq n$, $\hat{e}_i \in p_i$ and Π is a complete path in M.

Note that by construction a relational model M for a log L accepts all traces in L. To see this, consider a trace $l = \langle START, \hat{e}_1, \ldots, \hat{e}_n, END\rangle \in L$. The "next" relation, corresponding to R_{next}, holds for all pairs of adjacent event instances, that is $\forall 1 \leq i < n, \langle \hat{e}_i, \hat{e}_{i+1}\rangle \in R_{next}$ as well as $\langle START, \hat{e}_1\rangle \in R_{next}$ and $\langle \hat{e}_n, END\rangle \in R_{next}$. This means that l maps to a complete path in M and therefore M accepts l.

3.2 Invariants

We consider three invariants that relate pairs of event types:

Definition 10 (Event Invariant)**.** Let a and b be two event types. Then an event invariant is a property that relates a and b in one of the following three ways:

$a \rightarrow b$ **:** In a model, if some partition on a complete path contains an event instance of type a, then at least one later partition along that path contains an event instance of type b.

$a \not\rightarrow b$ **:** In a model, if some partition on a complete path contains an event instance of type a, no later partition along that path contains an event instance of type b.

$a \leftarrow b$ **:** In a model, if some partition on a complete path contains an event instance of type b, at least one earlier partition along that path contains an event instance of type a.

Definition 11 (Invariant Satisfiability)**.** Let M be a relational model, and let i be an event invariant. M satisfies i iff $\forall$ Π, a complete path in M, i is true of Π.

Each of the three event invariants may relate any pair of event types. Thus, for a set of event types E there can be at most $3|E|^2$ invariants.

Synoptic mines the above invariants by collecting three kinds of counts across all the traces. Each trace is traversed once in the forward and once in the reverse direction to count:

- $\forall a$,Occurrences$[a]$: the number of event instances of type a,
- $\forall a,b$ Follows$[a][b]$: the number of event instances of type a that are followed by at least one event instance of type b, and
- $\forall a,b$ Precedes$[a][b]$: the number of event instances of type b that are preceded by at least one event instance of type a.

The invariants are then determined by using the following equivalences:

$$\begin{aligned} a \rightarrow b &\Leftrightarrow \text{Follows}[a][b] = \text{Occurrences}[a] \\ a \not\rightarrow b &\Leftrightarrow \text{Follows}[a][b] = 0 \\ a \leftarrow b &\Leftrightarrow \text{Precedes}[a][b] = \text{Occurrences}[b] \end{aligned}$$

3.3 BisimH termination

This section proves that BisimH terminates (Theorem 1) and that the final model satisfies all the event invariants in the log (Theorem 2) and no others (Theorem 3).

Theorem 1 (BisimH Terminates)**.** *BisimH makes a finite number of iterations (no more than the number of event instances).*

Proof of Theorem 1. After every BisimH iteration, the model has more partitions than the model before the iteration, as some partition is split into two new partitions. The maximal model has a finite number of partitions (each event instance is in its own partition, except the *START* and *END* instances) and satisfies all the mined invariants. Therefore, BisimH can make no more iterations than there are event instances. □

Theorem 2 (True Invariant Satisfiability)**.** *BisimH produces a final model that satisfies all the event invariants that are true for a log.*

Proof of Theorem 2. The termination condition for BisimH is that the final model satisfies all the invariants mined from the log. The maximal model trivially satisfies all those invariants. Therefore, BisimH either terminates with that model or a smaller model that also satisfies the invariants. □

Theorem 3 (False Invariant Unsatisfiability)**.** *Let L be a log with event instances $\hat{E}$ and event types E. Let $M = \langle M_V, M_A \rangle$ be a relational model over $\hat{E}$ with a family of relations R_r. Let i be an event invariant that is not true of L. Then M does not satisfy i.*

Proof of Theorem 3. Since i is not true for L, there must be a trace $l \in L$ for which i is not true. Since there exists a path in M corresponding to l, that path must start with *START* and end with *END* and must not satisfy i. Therefore, M does not satisfy i. □

3.4 Improving model search efficiency

The previous section proved that regardless of which partitions BisimH splits, the algorithm always finds a model that satisfies exactly the log invariants. This is an important theoretical result, but a splitting strategy must be efficient in practice since refinement is expensive — a log with dozens of event types will generally satisfy hundreds of invariants of the types we are considering. Checking these invariants is costly, especially when the model grows to a large size. In this section, we prove that once BisimH satisfies an invariant, it never again violates it (Theorem 4). Therefore, invariants that have been satisfied do not need to be re-checked in finer models. This reduces the number of model checking runs BisimH needs to perform, making it more efficient.

Theorem 4 (Invariant Preservation)**.** *Let L be a log with event instances $\hat{E}$ and event types E. Let $M = \langle M_V, M_A \rangle$ be a relational model over $\hat{E}$ with a family of relations R_r. Construct a new relational model $M' = \langle M'_V, M'_A \rangle$ as follows:*

1. *Select one partition $p \in M_V$, $|p| \geq 2$.*
2. *Split p into two nonempty partitions p' and p''.*
3. *Let $M'_V = (M_V \setminus \{p\}) \cup \{p', p''\}$.*
4. *Compute M'_A using the family of relations R_r.*

For all event invariants i:

$$[(M \text{ satisfies } i) \wedge (i \text{ is true for } L)] \Rightarrow M' \text{ satisfies } i.$$

It is an immediate corollary that M' satisfies all invariants that M does and that are true in the log.

Proof sketch: The proof considers the differences between M and M', and relies on the fact that these differences are confined to the region around the refined partition p. Consider some path Π' in M' that might violate invariant i. That path is made up of edges, each of which comes from some trace, which means for each edge, there is a corresponding edge in M. Therefore, there is a corresponding path Π in M that goes through partitions of the same event types. Since the event types along both paths are identical, then either both or none of the paths satisfy i. But since M satisfies i, so must M'.

Proof of Theorem 4. Without loss of generality, let $\langle a, b \rangle$ be the pair of event types that i relates. We will now show that all paths in M' must satisfy i.

Consider a complete path Π' in M'. For any two partitions connected by an edge in Π' there must exist at least one pair of event instances, one in each partition, that is related by some relation in some trace. For each edge in Π' choose such a pair of event instances to construct a sequence of event instances s.

Now consider the unique path Π in M that corresponds to s. Every partition in Π contains event instances of the same type as the corresponding partition in Π' (in fact, Π's partition is a superset of Π''s).

Assume Π' violates i. Consider three cases:

Case 1: i is $a \rightarrow b$. There must be some partition in Π' with an event instance of type a such that no subsequent partition in Π' contains an event instance of type b. Therefore, no subsequent partition in Π contains an event instance of type b. But M satisfies i. Contradiction.

Case 2: i is $a \not\rightarrow b$. There must be some partition in Π' with an event instance of type b that follows a partition with an event instance of type a. Then Π must also violate i. Contradiction.

Case 3: i is $a \leftarrow b$. There must be some partition in Π' with an event instance of type b such that no earlier partition in Π' contains an event instance of type a. Then no earlier partition in Π contains an event instance of type a. But M satisfies i. Contradiction. □

3.5 Maintaining a small model size

Synoptic's aim is to present to a developer the smallest model (fewest nodes) satisfying the mined invariants. Large models are often too complex and no better than the raw log. In this section, we explain how the CEGAR [7] approach leads BisimH towards concise models. We argue that refinement always makes provable progress towards satisfying an invariant. Therefore BisimH rarely performs splits that make the model larger than it needs to be.

As a reminder, the CEGAR approach (detailed in Section 2.4.1) works as follows. First, BisimH generates a counterexample trace (e.g., Figure 5) that is accepted by the model and violates a mined invariant (*valid-coupon* $\not\rightarrow$ *invalid-coupon*). BisimH then traces along the counterexample trace in the model and in the input traces to find the longest prefix of partitions that exists as a sequence of corresponding event type instances in at least one input trace. The last partition in this prefix is refined (Figure 6). Two cases are possible:

Case 1: The refined model does not accept the counterexample trace. Consider the set of trace equivalence classes: two traces are in the same class if the paths of the two traces in the model are equivalent after removing all iterations through loops in the model. A split that eliminates one loop-free trace from an equivalence class, eliminates all traces in that class. Thus, eliminating a counterexample always eliminates an entire class of counterexamples that violate the invariant. Since there are a finite number of loop-free paths in a model, eliminating a class makes progress toward satisfying the invariant.

Case 2: The refined model accepts the counterexample trace. Consider the prefix corresponding to the counterexample trace in the refined model. This prefix is shorter than the previous prefix by at least one partition (the partition that was refined). Because any prefix must be finite, the refinement makes progress toward eliminating the counterexample trace from the model (towards Case 1).

Because BisimH considers one counterexample at a time, a refinement may split a partition suboptimally. That is, a split partition may need to be split again to help eliminate another counterexample, even though a single split might suffice to help eliminate both counterexamples. To counteract such suboptimal splits, BisimH uses coarsening (Section 2.4.2).

4. Experience with Synoptic

We performed two case studies to evaluate Synoptic's ability to produce concise and useful representations in practice. First, we carried out a user study with a developer working on the *reverse traceroute* system that determines the likely reverse route from an arbitrary destination on the Internet to a source host [23] (Section 4.1). Synoptic analyzed the coordinator node logs that contained debugging event instances generated by the system.

Second, we introduced Synoptic as a tool for use in an undergraduate distributed systems class of 45 students (Section 4.2). The students were tasked with designing and implementing a cache coherence protocol and had to (1) draw a finite state machine of their design, (2) run Synoptic on their implementation, and (3) explain any observed differences.

The students used Synoptic during development and testing, while the reverse traceroute developer used Synoptic on logs generated in production. We therefore believe that Synoptic can be helpful during all stages of a typical software engineering process. Overall, we found that Synoptic was useful for finding new bugs (Section 4.3), for increasing developer confidence (Section 4.4), and for building understanding (Section 4.5).

4.1 Reverse traceroute study

Reverse traceroute [23] is a distributed system that determines the likely reverse traceroute from an arbitrary destination on the Internet to a source host. Reverse traceroute relies on a distributed set of Internet vantage points hosted by PlanetLab [36], and uses a variety of methods to find each segment of the reverse route, such as IP record route and timestamp options [19, 20], and relies on IP spoofing from PlanetLab hosts.

Reverse traceroute has been in live deployment for over 7 months and since that time it has had hundreds of distinct users and has handled a total of 3.6 million requests. Today, it gets tens of thousands of requests per day. Recently, a large, popular, and ubiquitous Internet company has deployed the system internally.

We carried out a user study with a developer working on the system to study a log of 900,000 event instances. To generate this log, the developer spent a total of 15 minutes to add a total of 16 lines of logging code to the system. We then wrote four regular expressions to process the log. The log was divided into traces by measurement-based method names — a single trace corresponded to a sequence of method calls made to determine the reverse route for a particular ⟨source, destination⟩ pair.

Synoptic took 11.5 minutes to generate the final graph from the input log on an Intel i7 (2.8 GHz) OS X machine with 8GB of RAM. Because this graph contained many rare edges (i.e., edges with low transition probabilities), we showed the developer both the full graph, as well as a graph that omitted 62 edges with low transition probability. The second type of graph is shown in Figure 7. We then performed a talk-aloud user study with the developer by showing him Synoptic-derived graphs, explaining to him what they represent, and asking him to talk through his observations.

4.2 Distributed systems course assignment

In the University of Washington undergraduate distributed systems course[2], groups of 2–4 students designed and implemented a peer-to-peer Facebook-like social network. The project was divided into multiple assignments, one of which was to implement the distributed version of a cache coherence protocol [26] between a single master node and some number of replica clients. For this assignment, the students were to (1) record their design as a FSM diagram, (2) implement their design, (3) apply Synoptic to logs generated by their implementation, and (4) observe and explain any differences between the Synoptic output and their initial FSM diagram.

For testing, the students used a simulated environment in which all nodes executed in a single process, and communicated via a centralized simulator manager. The simulator provides the option of reordering, losing, and duplicating messages, as well as randomly failing and restarting nodes.

The simulator logged common event types like message sent, received, and lost and file read and written, and also allowed student node code to log user-defined event types. Although the simulated system was distributed, the simulator produced totally ordered logs — event instances were serialized through the central simulator manager. The students were also given a set of Synoptic regular expressions for processing logs generated by the simulator.

All 18 groups completed the assignment. Due to space constraints, Sections 4.3–4.5 showcase just a few of the Synoptic diagrams generated by the students, and quotes just a few of their reports on their experiences with Synoptic.

[2]http://www.cs.washington.edu/education/courses/cse490h/11wi

4.3 Finding bugs in code

Synoptic models capture event type orderings and co-occurrence frequencies among event types. The absence of an edge could indicate that the log is incomplete. However, if the behavior is supposed to occur at all times or with high frequency, an unexpected graph topology can be an indicator of a latent bug.

Reverse traceroute study.

The reverse traceroute developer identified one new and important bug using the Synoptic model within the first two minutes of seeing the model. All measurements made by the system must eventually terminate in either the `do_reach_callback` or the `do_fail_callback` methods. The developer thought that all traces reaching these methods terminated. The graph showed otherwise — some of the traces continued past these callbacks. The model in Figure 7 illustrates these buggy transitions with bold, dashed emphasis. The developer hypothesized that this bug is caused by concurrency in the measurement code. The developer also observed that the tool offers a light-weight means of verifying that some previously observed buggy behavior is not present after a bug fix, and that it may help to rule out bug fixes that fail to eliminate buggy behavior.

Distributed systems course.

Of the 18 groups, 3 groups found bugs in their implementations with Synoptic. Synoptic models effectively capture event type orderings and all three of the bugs had to do with illegal message orderings. One group observed that a transition that was expected never occurred — the node seemed to never execute the write command after processing it. They then fixed the bug and used Synoptic to confirm that it did not appear in the traces:

> "We did find a bug in the graph. If you follow the append path in the final graph you can see that it goes from append→send→write. In the old graph the append→send, but dies instead of passing it onto write."

A different group found a bug in which they mistakenly sent the wrong type of packet:

> "We had few places where we sent the wrong type of packet in the code. For example, we sent RDC when we had to send WDC. When looking at the Synoptic diagram, these kinds of mistakes were easy to find."

4.4 Increasing developer confidence

Synoptic models can succinctly represent thousands of execution traces with a few nodes in a graph. A single compact diagram that consolidates many executions gives developers confidence that they will not overlook any behaviors present in the log. Moreover,

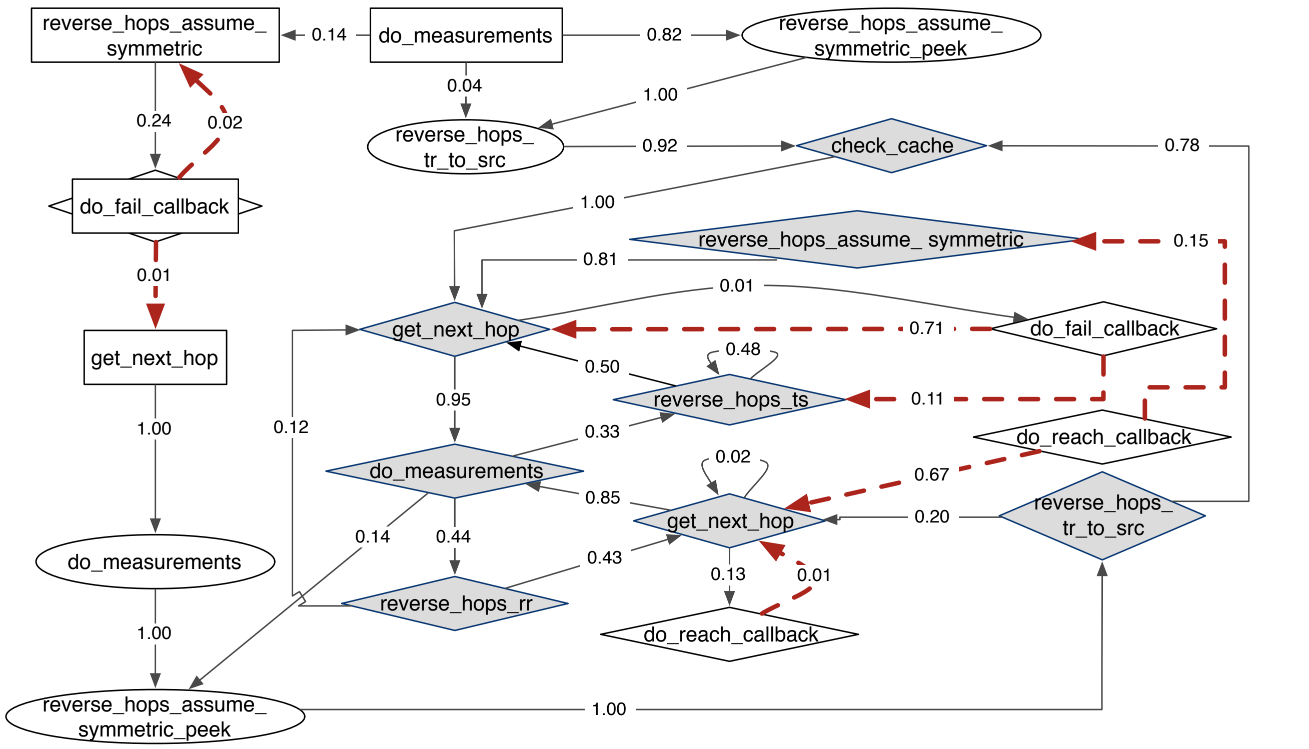


Figure 7: Synoptic model for a reverse traceroute input log of 900,000 event instances. Rectangular nodes are start nodes, and diamond nodes are terminal nodes. Edge labels indicate transition probability. For clarity certain edges and nodes are omitted. The (manually) bold, dashed lines indicate a new bug that was discovered by the developer. The (manually) shaded terminal nodes make up the set of methods exhibiting a bug known to the developer. Before viewing the Synoptic graph, the developer thought the bug affected only two of these eight nodes.

developers often recognized expected patterns and considered them to be important evidence that the system worked as expected.

Reverse traceroute study.

The reverse traceroute developer was often prompted by the model to try to explain various patterns. Patterns that were simpler and more noticeable, like self-loops on nodes, elicited more attention. For example, upon noticing a self-loop on one of the nodes the developer mentioned that this indicated that a specific type of measurement was re-tried and that this was correct behavior.

Distributed systems course.

Of the 18 groups in the class, 11 reported that Synoptic increased their confidence in their implementations. In many cases the students recognized expected patterns. Figure 8 illustrates two sets of diagrams generated by a group that felt that they acquired additional confidence in their system by using Synoptic. The figure shows four models, with each pair corresponding to a *Client* and *Server* processes in the system. The group decided to use a combination of messages and states for their event types — e.g., `create` is a file create request message sent by the client to the server, while e.g., `readonly_state` is the state of the client when it holds a shared read lock on the file. To more easily follow the sequence of event types the students generated two sets of models for two distinct scenarios (see Figure 8). By inspecting these models, the group confirmed that the messages were exchanged in the appropriate order and that the nodes transitioned between states correctly.

The following are some student quotes that indicate that Synoptic increased developers' confidence in their systems:

"[Synoptic] definitely let us know for sure that our code was functioning correctly."

"Using Synoptic did not help us find any bugs with our code, but it did help us to clarify that our code is doing what it should."

"We can confidently say that Synoptic helped confirm the correct behavior of our program, and certainly made us feel better about our code."

4.5 Building system understanding

Using the Synoptic model, the reverse traceroute developer was able to solidify his understanding about the system. For example, he knew that the system had a bug in which a reverse traceroute measurement terminates prematurely. Using the model, he was able to verify that this bug occurred — methods terminating prematurely appeared as terminal nodes in the model. However, as it turned out, the developer did not understand the full extent of this bug. He assumed that it affected only two methods. By inspecting the model, he found out that other methods were impacted as well. The model in Figure 7 illustrates the set of all the methods impacted by this known bug with a darker shading. This experience solidified the developer's understanding of where the bug manifested and he felt better prepared to resolve it.

Overall we found that Synoptic was useful for finding new bugs, for increasing developer confidence, and for building systems understanding.

4.6 Threats to validity

Our two user studies are limited in scope and have a number of inherent biases for which we were unable to control. The reverse traceroute system has been developed by about five developers, all of whom understand the entire system. Consequently, Synoptic models are straightforward for them to interpret. Developers who are new to a project or are working on a larger project may find it

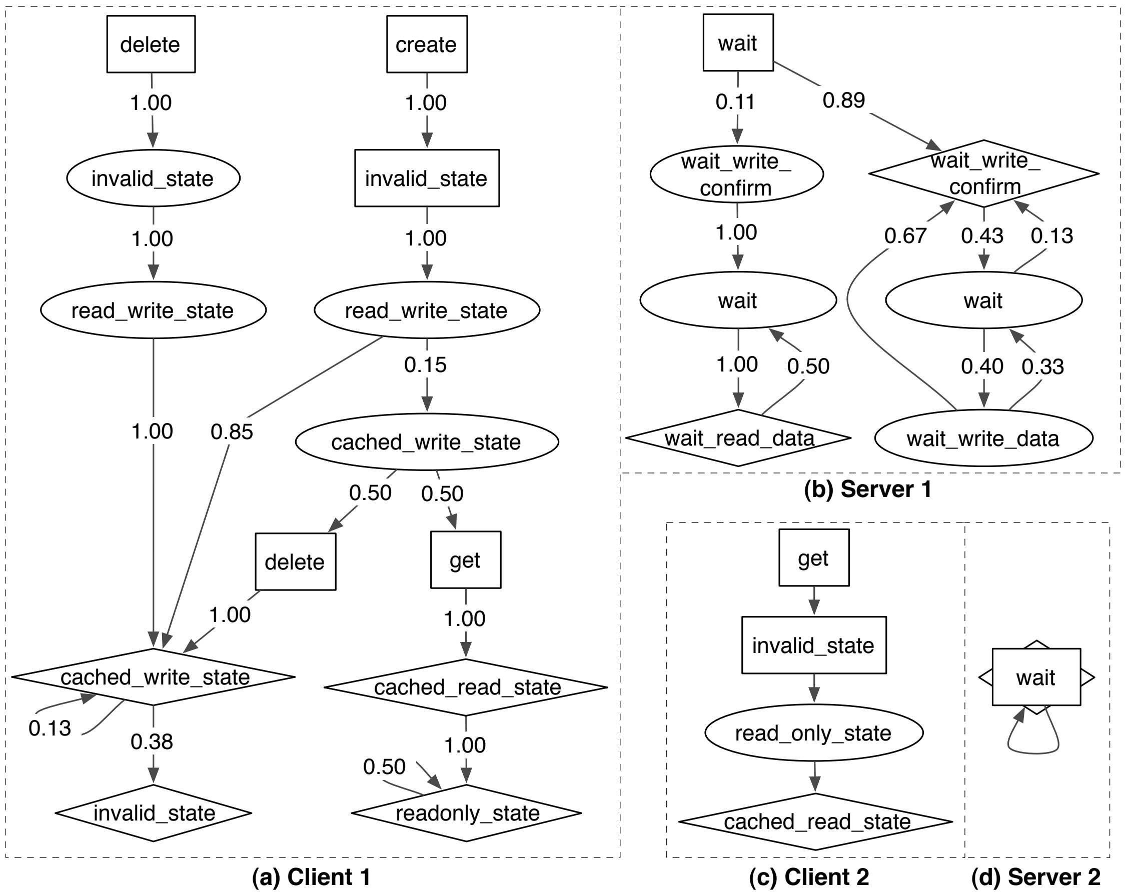


Figure 8: Two pairs of Synoptic models generated by a group of students in the distributed systems class for a distributed cache coherence assignment. Each pair of models has a server model and a client model. The **Client1** and **Server1** models correspond to a scenario in which the client host starts, and then deletes the file if it exists. Alternatively the client creates the file, and then either deletes it or reads from it. The **Client2** and **Server2** models correspond to the simpler scenario in which the client reads a file that is not currently accessed by any other client.

difficult to interpret Synoptic models, which may be larger and more complex. However, we believe that Synoptic may also be used to gain insight into components of larger systems, and because most sizable systems are developed in a modular fashion, there may still be value in using Synoptic in large projects. Lastly, our study with the developer implicitly emphasized bug findings, which may have primed the developer into thinking more about bugs. In a different context, he might have been less successful in identifying bugs.

Because the students in the distributed systems course were required to use Synoptic as part of the assignment, it is unknown whether they would have been motivated enough to learn about and use the tool without a mandate. Students might have also been attempting to please us and thereby reported only positive experiences with the tool. Finally, students are not representative of experienced developers and we do not know whether the bugs they found using Synoptic are problems for expert developers.

5. Related work

Work related to Synoptic falls into three main categories; (1) tools to mine logs generated by systems; (2) algorithms to create concise models of system executions; and (3) the study of bisimulations, which motivated our development of BisimH.

Mining systems logs

This paper extends our previous work [39] with a formal analysis of the BisimH algorithm and a report on user experiences with the Synoptic tool. An overview of the Synoptic tool from a user's perspective is given in [4]. Other prior work on mining systems logs focused on detecting dependencies [31], anomalies [22, 32, 46, 49], and performance debugging [40, 41]. That work does not target the problem of finding a concise model for an arbitrary system generating the log. For instance, SALSA [40] and Mochi [41] extract and visualize node behavior of Hadoop [17] node logs to support performance debugging. This line of work is MapReduce-specific. Perracotta [48] mines and visualizes temporal properties of event traces, and it has been used to study program evolution [47]. Unlike Synoptic, Perracotta does not use the mined temporal properties to infer a model of the system.

Inferring models

The problem of automata inference from positive examples of executions is computable [6], but is NP-complete [16, 3], and the FSA cannot be approximated by any polynomial-time algorithm [35].

The kTail algorithm [5], used extensively in related work, takes a finite state model and produces a more compact one by recursively merging states whose root subgraphs are identical up to a depth of *k*. Approaches that leverage kTail to infer models without developer supervision [1, 5, 27, 28, 30, 33, 37] can produce precise models for small and simple systems, but when complexity of the system increases, the precision of the inferred models decreases dramatically [27]. Lorenzoli et al. [30] developed GK-Tail, a variant of kTail, and applied it to logged sequences of method call invocations. Unlike BisimH, the GK-Tail algorithm does not preserve trace invariants. Lo et al. [29] augment the kTail algorithm by using temporal properties mined from execution traces to guide state merging while ensuring that the final model satisfies temporal constraints. Temporal-invariant-consistency greatly increases the model's precision. Synoptic produces similar high-precision models while leveraging refinement, as opposed to coarsening, to greatly increase the efficiency and scalability of the approach [39]. Krka et al. [25] have proposed, though have not yet implemented, using refinement and mined invariants to improve precision of inferred models beyond that of Lo et al.'s approach.

Numerous techniques leverage developer-written specifications to infer system models. Whittle and Schumann [44] generate component statecharts from scenarios and properties. Damas et al. [8] inductively infer labeled transition system (LTS) models from scenarios interactively provided by the developer. A later extension of this approach reduces the number of questions to the developer [9] by incorporating FLTL properties [15]. However, these techniques can synthesize overspecified models and require significant human input. Uchitel et al. [43] proposed using message sequence charts [21] to infer LTS models and discover implied scenarios. Harel et al. [18] synthesize statecharts from live sequence charts [10]. LTSs can also be constructed based on pre- and postcondition specifications [2, 11]. De Caso et al. [11] generate abstract models to support validation of the specifications. Alarjeh et al.'s technique [2] facilitates refinement of pre- and postconditions based on system goals and execution scenarios. Similarly, Krka et al.'s algorithm [24] synthesizes behavioral models from pre- and postcondition specifications and execution scenarios and can synthesize component-level models from system-level specification. Uchitel et al. [42] argued that it is crucial to consider the specifications' partiality when using developer-written specifications to infer models. In contrast to all these approaches, Synoptic requires much less input from the developer — the logs that are usually already generated by systems, and a small set of regular expressions. However, systems that are not instrumented to generate logs may require developers to change the implementation. However, logging is considered to be generally useful and adding such instrumentation leads to better software. Further, we hope that Synoptic's utility will motivate developers to increase their systems' logging capabilities.

Bisimulation

A bisimulation is a simulation relation that provides a strong notion of similarity for relational structures [38]. Its key feature is to preserve certain properties of the relational structure, for example, two strongly bisimilar transition systems are guaranteed to satisfy the same set of LTL formulae. An important application in model checking is model minimization [14]. Our BisimH algorithm is a modification of a partition refinement algorithm [34], 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 BisimH algorithm is also related to the partition refinement algorithms in [13], but BisimH uses invariants to guide exploration and termination.

6. Limitations and future work

While working on Synoptic, we observed a number of its limitations. Here, we detail the most important of these and connect some of them to our future work.

Applicability. Synoptic models capture ordering relationships between events observed in a log. It does not handle algebraic and logical relationships that may also be useful in modeling software (e.g., this.next $\neq$ this.prev). Synoptic is therefore best suited for studying logs of systems whose execution can be modeled as a sequence of elements, with the ordering, the presence, and absence of elements encoding some useful semantics about the system. We have observed that Synoptic can help with problems whose root causes can be deciphered using such semantic information. More advanced issues, however, would require richer and more complex models than Synoptic currently provides.

Invariants. Synoptic relies on three temporal invariants to determine when to terminate and how to proceed during refinement. A rigorous evaluation of the limitations and advantages of these invariant types is necessary. For example, we know that the invariants constrain Synoptic models in ways that are sometimes undesirable. For instance, the $\not\rightarrow$ invariant constrains Synoptic models to be less generative: e.g., if $a \not\rightarrow b$ is true, then the model is restricted from generating a path between a and b, even though this behavior might be valid and can appear in an execution that is not present in the input log. However, we do not know what kinds of systems or uses these invariants favor, and whether we should expand this set, or make it smaller.

Synoptic invariants are temporal. They do not involve the *data values* that are often present in logs. Extending Synoptic to mine and then preserve value-based invariants is a part of our future work.

Reliance on logs. To work well, Synoptic needs the input log to include as many different system executions as possible. This is because Synoptic models are at most as detailed as the input logs. If the user failed to log an important behavior, then this behavior will usually not be present in the Synoptic-generated model. However, generating all possible system behaviors is notoriously difficult, and may be infeasible, as illustrated by the following student quote:

"However, we had to run specific simulation cases in order to produce the log, so while Synoptic was very useful, most of the debugging process involved trying commands in the simulator. We knew what cases we were testing, so running them through the terminal was an easier way to test for the bug. But Synoptic did confirm that we have the right message flows."

Handling concurrency. Synoptic cannot handle traces of concurrent systems as it assumes a totally ordered relation for ordering event instances in a trace. Concurrent systems may be modeled by Synoptic with explicit concurrency (e.g., by listing all possible event instance permutations), but this results in highly connected models that are difficult to interpret. Another alternative, which we are currently pursuing, is to extend Synoptic models and the semantics of the three invariants types to accommodate partially ordered traces.

Fault localization/interactive tool support. The feasibility of fault localization using Synoptic-generated models depends on the density and quality of the logging statements. Reconstructing execution paths based on logs is an active research area (e.g., SherLog [49]), and we hope to leverage this existing work in developing more automated fault localization techniques. However, fault localization fundamentally requires human insight. To this end, we are working on a Synoptic GUI that will support developers in this task. This GUI will allow developers to explore, query, and interact with the Synoptic models in real-time. For example, developers will be able to find out which of the logged traces pass through a set of partitions in the model and which event instances belong to a partition, as well as other information.

7. Conclusion

Logging is a popular debugging methodology. Unfortunately, large logs are often complex and difficult to analyze manually. This paper presented the design and evaluation of a tool called Synoptic, which builds a system model from its execution logs. Unlike other tools, Synoptic requires few inputs from the developer and can be applied to pre-existing logs.

The key to Synoptic's algorithm is its use of three types of mined temporal invariants to guide the model space exploration. Our formal evaluation showed that Synoptic's algorithm always makes progress and always finds a model that satisfies the mined invariants. Our case studies showed that Synoptic graphs improved developer confidence in the correctness of their systems, and were useful for finding bugs.

We believe that Synoptic bridges the gap between systems developed by developers with little to no training in formal methods, and a suite of methods developed by the formal methods community. Synoptic is an open-source tool (`http://synoptic.googlecode.com`) and has met the expectations of FSE artifact evaluation committee.

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. This work is also supported by the Saarbrücken Graduate School of Computer Science, which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *Proc. of FSE*, 2007.

[2] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning Operational Requirements from Goal Models. In *Proc. of ICSE*, 2009.

[3] D. Angluin. Finding Patterns Common to a Set of Strings. *Journal of Computer and System Sciences*, 21(1):46 – 62, 1980.

[4] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst. Synoptic: Studying Logged Behavior with Inferred Models. In *Proc. of FSE*, 2011.

[5] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput.*, 21(6):592–597, 1972.

[6] L. Blum and M. Blum. Toward a Mathematical Theory of Inductive Inference. *Information and Control*, 28(2):125 – 155, 1975.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *Computer Aided Verification*, pages 154–169. Springer, 2000.

[8] C. Damas et al. Generating Annotated Behavior Models from End-User Scenarios. *IEEE TSE*, 31(12), 2005.

[9] C. Damas, B. Lambeau, and A. van Lamsweerde. Scenarios, Goals, and State Machines: a Win-Win Partnership for Model Synthesis. In *Proc. of FSE*, 2006.

[10] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Form. Meth. Syst. Des.*, 19(1), 2001.

[11] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Validation of Contracts Using Enabledness Preserving Finite State Abstractions. In *Proc. of ICSE*, 2009.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. of ICSE*, 1999.

[13] T. Elomaa. Partition-Refining Algorithms for Learning Finite State Automata. In *Proc. of ISMIS*, 2002.

[14] K. Fisler and M. Y. Vardi. Bisimulation Minimization and Symbolic Model Checking. *Formal Methods in System Design*, 21(1):39–78, 2002.

[15] D. Giannakopoulou and J. Magee. Fluent Model Checking for Event-Based Systems. In *Proc. of FSE*, 2003.

[16] E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.

[17] Welcome to Apache Hadoop!, `http://hadoop.apache.org/`. Accessed March 9, 2011.

[18] D. Harel, H. Kugler, and A. Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. *Form. Meth. in Soft. and Sys. Modeling*, 3393, 2005.

[19] IPv4 Specification, Record Route option. `http://www.ietf.org/rfc/rfc791.txt`. Pg. 20, 21. Accessed March 9, 2011.

[20] IPv4 Specification, Timestamp option. `http://www.ietf.org/rfc/rfc791.txt`. Pg. 22, 23. Accessed March 9, 2011.

[21] ITU. Message Sequence Charts, 2000.

[22] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. In *Proc. of ICAC*, 2005.

[23] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. van Wesep, T. Anderson, and A. Krishnamurthy. Reverse Traceroute. In *Proc. of NSDI*, 2010.

[24] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic. Synthesizing Partial Component-Level Behavior Models from System Specifications. In *Proc. of FSE*, 2009.

[25] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using Dynamic Execution Traces and Program Invariants to Enhance Behavioral Model Inference. In *Proc. of ICSE*, 2010.

[26] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, 7:321–359, November 1989.

[27] D. Lo and S.-C. Khoo. QUARK: Empirical Assessment of Automaton-based Specification Miners. In *Proc. of WCRE*, 2006.

[28] D. Lo and S.-C. Khoo. SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner. In *Proc. of FSE*, 2006.

[29] D. Lo, L. e. Mariani, and M. Pezzè. Automatic Steering of Behavioral Model Inference. In *Proc. of FSE*, 2009.

[30] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *Proc. of ICSE*, 2008.

[31] J.-G. Lou, Q. Fu, Y. Wang, and J. Li. Mining Dependency in Distributed Systems through Unstructured Logs Analysis. *SIGOPS Oper. Syst. Rev.*, 44:91–96, March 2010.

[32] J. G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining Invariants from Console Logs for System Problem Detection. In *Proc. of ATC*, 2010.

[33] L. Mariani and M. Pezzè. Dynamic Detection of COTS Component Incompatibility. *IEEE Software*, 24(5):76–85, September/October 2007.

[34] R. Paige and R. E. Tarjan. Three Partition Refinement Algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[35] L. Pitt and M. K. Warmuth. The Minimum Consistent DFA Problem Cannot be Approximated Within any Polynomial. *J. ACM*, 40(1):95–142, 1993.

[36] PlanetLab | An open platform for developing, deploying, and accessing planetary-scale services, `https://www.planet-lab.org`. Accessed March 9, 2011.

[37] S. P. Reiss and M. Renieris. Encoding Program Executions. In *Proc. of ICSE*, 2001.

[38] D. Sangiorgi. On the Origins of Bisimulation and Coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):1–41, 2009.

[39] S. Schneider, I. Beschastnikh, S. Chernyak, M. D. Ernst, and Y. Brun. Synoptic: Summarizing System Logs with Refinement. In *Proc. of SLAML*, 2010.

[40] J. Tan, X. Pan, S. Kavulya, R. G, and P. Narasimhan. SALSA: Analyzing Logs as StAte Machines. In *Proc. of WASL*, 2008.

[41] J. Tan, X. Pan, S. Kavulya, R. G, and P. Narasimhan. Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop. In *Proc. of WASL*, 2009.

[42] S. Uchitel, J. Kramer, and J. Magee. Behaviour Model Elaboration Using Partial Labelled Transition Systems. In *Proc. of FSE*, 2003.

[43] S. Uchitel, J. Kramer, and J. Magee. Incremental Elaboration of Scenario-Based Specifications and Behavior Models Using Implied Scenarios. *ACM TOSEM*, 13(1), 2004.

[44] J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Proc. of ICSE*, 2000.

[45] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Experience Mining Google's Production Console Logs. In *Proc. of SLAML*, 2010.

[46] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *Proc. of SOSP*, 2009.

[47] J. Yang and D. Evans. Automatically Inferring Temporal Properties for Program Evolution. In *Proc. of ISSRE*, 2004.

[48] J. Yang and D. Evans. Dynamically Inferring Temporal Properties. In *Proc. of PASTE*, 2004.

[49] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *Proc. of ASPLOS*, 2010.