Synoptic: Studying Logged Behavior with Inferred Models

Ivan Beschastnikh Jenny Abrahamson Yuriy Brun Michael D. Ernst

Computer Science & Engineering
University of Washington
{ivan,jabrah,brun,mernst}@cs.washington.edu

Abstract

Logging is a powerful method for capturing program activity and state during an execution. However, log inspection remains a tedious activity, with developers often piecing together what went on from multiple log lines and across many files. This paper describes Synoptic, a tool that takes logs as input and outputs a finite state machine that models the process generating the logs. The paper overviews the model inference algorithms. Then, it describes the Synoptic tool, which is designed to support a rich log exploration workflow.

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

Examining logs is one of the most popular means of gaining insight into program execution. Developers add logging statements to record events and state transitions. The resulting logs are then inspected to find anomalies, verify correctness, debug performance, and for other tasks.

Unfortunately, developers find it difficult to inspect and reason about logged information. As an example of this consider Figure 1, which lists two log snippets based on a real log of security-related events on an OS X system. Each snippet represents a sequence of login attempts resulting in authorization. One of the two snippets contains a bug, but it is difficult to tell which one.

This paper describes Synoptic[1] — a tool that takes a log as input and infers a compact finite state machine model of the process that generated the log. Previous work has referred to this procedure as model inference, specification mining, and process discovery. For example, a longer log consisting of login attempts like the ones listed in Figure 1 can be processed with Synoptic to derive the model in Figure 4(b). This model captures the essential information necessary to understand basic temporal relationships between the logged events and can be used for various tasks by the developer. For example, this model makes it easier to notice the aforementioned bug — a failed authentication attempt sometimes results in an authorized login. The developer can also be more confident that the bug has been successfully removed by inspecting the model generated with Synoptic for a log of a system with the bug fix.

```
loginwindow[35]: Login Window Started Security Agent
May 20 16:15:27 my-mac SecurityAgent[130]: Showing Login Window
May 20 16:29:19 my-mac SecurityAgent[130]: User info context values set for jenny
May 20 16:29:19 my-mac authorizationhost[129]: Failed to authenticate user <jenny> (tDirStatus: -14090).
May 20 16:29:22 my-mac SecurityAgent[130]: User info context values set for jenny
May 20 16:29:22 my-mac SecurityAgent[130]: Login Window Showing Progress
May 20 16:29:22 my-mac SecurityAgent[130]: Login Window done
May 20 16:29:22 my-mac com.apple.SecurityServer[23]: Succeeded authorizing right
'system.login.console' by client '/System/Library/CoreServices/loginwindow.app' for authorization created
by '/System/Library/CoreServices/loginwindow.app'
```

(a)

```
loginwindow[35]: Login Window Started Security Agent
May 22 07:24:18 my-mac SecurityAgent[130]: Showing Login Window
May 22 07:25:13 my-mac SecurityAgent[130]: User info context values set for ivan
May 22 07:25:13 my-mac authorizationhost[129]: Failed to authenticate user <ivan> (tDirStatus: -14090).
May 22 07:25:15 my-mac SecurityAgent[130]: Login Window Showing Progress
May 22 07:25:15 my-mac SecurityAgent[130]: Login Window done
May 22 07:25:16 my-mac com.apple.SecurityServer[23]: Succeeded authorizing right
'system.login.console' by client '/System/Library/CoreServices/loginwindow.app' for authorization created
by '/System/Library/CoreServices/loginwindow.app'
```

(b)

Figure 1: Two log snippets based on the `/var/log/secure.log` file found in OS X 10.6.8. Each snippet represents a sequence of login attempts resulting in authorization. One of the snippets contains a security bug. Can you figure out which one? Answer is in the footnote.[2] Synoptic helps with the task of understanding what is in a log by generating a model that describes it (Figure 4(b)).

Synoptic works on logs of systems that can be modeled as a finite state machine. In particular, Synoptic cannot yet reason about concurrency. Further, the log must contain sufficient information to effectively model the system. The efficacy of a given model depends on its intended use. More concretely, Synoptic requires an input log that (1) contains one or more system executions, each of which can be thought of as a path through a finite state machine; (2) totally orders the log lines belonging to the same execution (e.g., with respect to a time field, or the ordering of lines in the file); and (3) captures the abstract event type or abstract state of the system on each line belonging to an execution.

We have previously formally evaluated Synoptic's algorithms and showed that Synoptic-generated models can help developers discover bugs [3]. In this paper, we briefly summarize Synoptic's model inference algorithm (Section 2). We then focus on the design of the Synoptic tool (Section 3) and describe how Synoptic supports a workflow in which developers spend most of their time analyzing the derived Synoptic models (Section 4). We end with a survey of related tools and techniques (Section 5).

2. How Synoptic works

Figure 2 summarizes how Synoptic works. This section overviews Synoptic's mechanisms by working through an example log based

[1] `http://synoptic.googlecode.com`

[2] The two snippets differ structurally in a single place — `User info context...` line appears twice in snippet (a) and once in snippet (b). Snippet (b) contains the bug since the user is authorized even though he failed to authenticate.

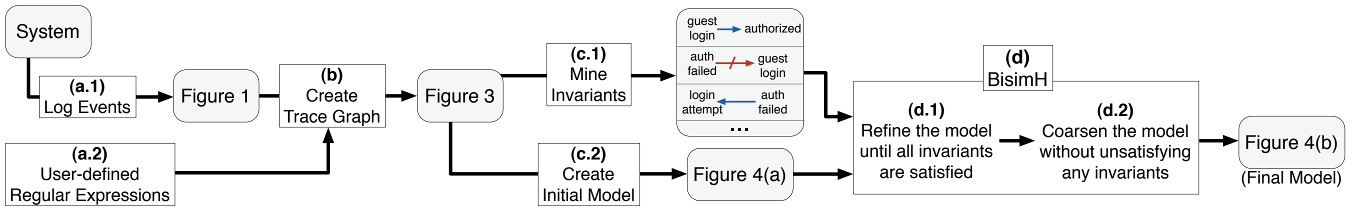

System
(a.1) Log Events
(a.2) User-defined Regular Expressions
Figure 1
(b) Create Trace Graph
Figure 3
(c.1) Mine Invariants
guest login → authorized
auth failed ↛ guest login
login attempt ← auth failed
...
(c.2) Create Initial Model
Figure 4(a)
(d) BisimH
(d.1) Refine the model until all invariants are satisfied
(d.2) Coarsen the model without unsatisfying any invariants
Figure 4(b)
(Final Model)

Figure 2: A step-by-step depiction of how Synoptic works.

on Figure 1. We refer the reader to [3] for a more detailed treatment of the algorithm, as well as proofs of several of its key properties.

2.1 Constructing the trace graph

Synoptic constructs a model from a set of system execution traces. It takes as input a log file containing the traces and a set of regular expressions. The log is parsed using the expressions to extract from some of the lines an *event instance*, a triplet of: (1) a trace identifier, (2) an optional timestamp, and (3) an *event type*. Trace identifiers group event instances into *traces* — linear graphs with vertices representing event instances and edges capturing their ordering. Synoptic requires event instances in a trace to be totally ordered. Timestamps may be used for this or the order could be derived implicitly from the order of lines in the log. Lastly, an event type is an arbitrary string, defined by the developer as something that conveys important information about the system.

The union of traces is a *trace graph*. Figure 3 shows a trace graph with five traces for a log based on Figure 1: a trace represents a series of login attempts ending with authorization. The traces are extracted with an implicit log line ordering and four regular expressions:

```
.+User info.+guest$(?<TYPE=>guest login)
.+User info.+(?<TYPE=>login attempt)
.+Failed to authenticate.+(?<TYPE=>failed auth)
.+Succeeded authorizing.+(?<TYPE=>authorized)
```

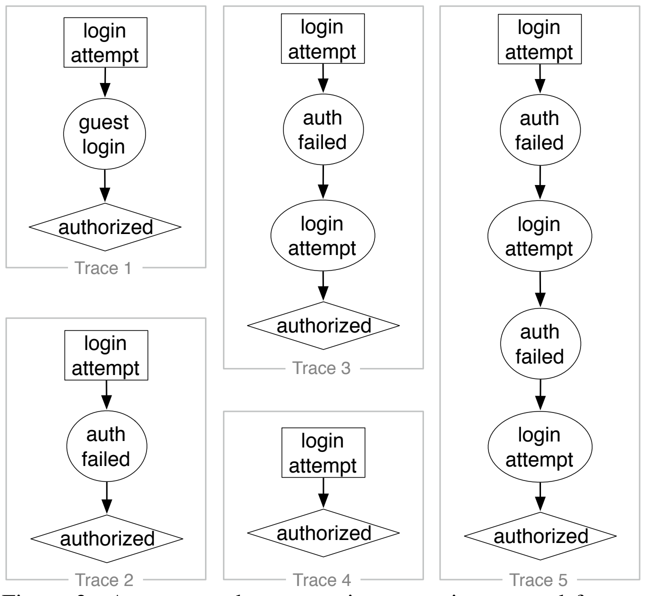


Figure 3: A trace graph representing executions parsed from a log based on Figure 1. Rectangular/diamond/oval nodes indicate initial/terminal/intermediate events. An execution represents some login attempts terminating in authorization. Trace 2 contains a bug.

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:

a Always Followed by b ($a \rightarrow b$). Whenever the event a appears, the event b must follow later in the same trace.

a Never Followed by b ($a \not\rightarrow b$). Whenever the event a appears, the event b never appears later in the same trace.

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

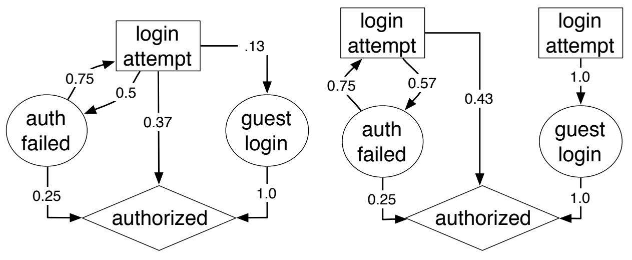
We term these relations "invariants" because they succinctly capture temporal event type relationships that hold true over all the input traces. These invariants are based on the most frequently observed specification patterns in Dwyer et al. [7]. In practice, we found these invariants to be sufficient for capturing key temporal properties of systems whose logs we've considered. The trace graph in Figure 3 yields 16 such invariants. One example is *auth failed* $\not\rightarrow$ *guest login*.

2.3 Synoptic models and the initial model

A Synoptic model is a *partition graph* of the trace graph. Given a partitioning of the original vertices with each partition containing event instances of the same event type, each vertex in the model represents one partition. A directed edge between two vertices indicates that there is a pair of event instances in the corresponding partitions that are adjacent with respect to the total order relation in at least one of the input traces. This model makes minimal assumptions about the underlying process that produced the logged events [9].

An important property of a Synoptic model is that each trace in the input log is accepted by a model constructed from the corresponding event instances (a 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.

Synoptic's core algorithm (BisimH) starts with an *initial model* in which there is one partition per event type containing all the event instances of that type. This is the most compact or abstract model. Figure 4(a) shows the initial model for the trace graph in Figure 3. BisimH then refines the initial graph until it satisfies all the invariants mined from the trace graph.



(a) Initial model (b) Final model

Figure 4: **(a)** Initial mode and **(b)** final model for the trace graph in Figure 3. Edge labels indicate transition probabilities. Note how the *login attempt* node in (a) is refined in (b).

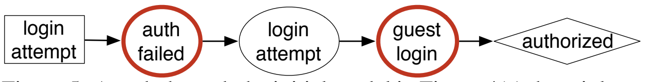


Figure 5: A path through the initial model in Figure 4(a) that violates the mined *auth failed* $\not\to$ *guest login* invariant.

2.4 Refining models to satisfy invariants

BisimH attempts to pick a minimal sequence of partition refinements (i.e., splits), to produce a smallest graph satisfying the mined invariants. This problem is NP-hard [5] so BisimH uses heuristics to produce a result that is good in practice.

BisimH performs refinement as long as some mined invariant is not satisfied by all paths in the model. It uses a model checker to check if a model satisfies a mined invariant. The checker produces a counterexample path when the model does not satisfy an invariant. For example, the invariant *auth failed* $\not\to$ *guest login* mined from the trace graph in Figure 3 is not satisfied by the initial model in Figure 4(a). Figure 5 shows the corresponding counterexample path.

Having identified a set of counterexamples that violate the mined invariants, BisimH follows the counterexample guided abstraction refinement approach [5] to identify a set of *candidate partitions*, for each of which there exists a split that removes at least one of the counterexamples. BisimH finds these partitions heuristically by tracing each counterexample, simultaneously in the initial traces and in the model. In the traces, only a prefix of the counterexample path will be present. 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 creating the counterexample path. The candidate partition is refined to eliminate the spurious transition. For example, to eliminate the counterexample path in Figure 5 from the model in Figure 4(a), the *context set* partition is split into the set of events that can and cannot reach any events in the *guest login* partition. Figure 4(b) shows the resulting refined model.

2.5 Compacting the model with coarsening

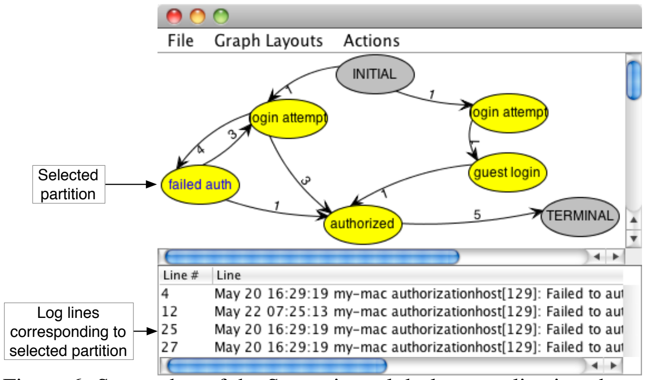
The BisimH algorithm may end up refining more than is necessary. When this happens, the model will contain partitions that can be merged without violating the satisfied invariants. After refinement, BisimH merges such partitions using *kTail-equivalence* [4] (k=0). The resulting merged model is locally minimal: merging any two partitions will violate some invariant. In the running example, no coarsening is necessary. Figure 4(b) is the final model.

3. Synoptic design

We designed Synoptic as a public web service that can be accessed with a browser, and also as a stand-alone application that can be downloaded to and run on a user's personal computer (both are available at `http://synoptic.googlecode.com`). These two approaches provide different trade-offs, and we support both for greater user flexibility.

Synoptic web service. A web service allows us to transparently update the code and to improve the user's experience without requiring users to download a new software version. Another important benefit is that we can transparently parallelize many of the Synoptic algorithms on the back-end, thus providing users with better performance than if Synoptic were to run on a single machine. Users can also more easily share Synoptic output with others (e.g., by sharing a URL). We also provide users with the option of downloading and running a Synoptic web service instance of their own.

Logs can be uploaded to the service as files. Or, if the developer is using log4j or log4net, the stream of logged messages can be directed to the web service using a TCP socket. The second option



```
4   May 20 16:29:19 my-mac authorizationhost[129]: Failed to aut
12  May 22 07:25:13 my-mac authorizationhost[129]: Failed to aut
25  May 20 16:29:19 my-mac authorizationhost[129]: Failed to aut
27  May 20 16:29:19 my-mac authorizationhost[129]: Failed to aut
```

Figure 6: Screenshot of the Synoptic tool desktop application showing the final model in Figure 4(b).

is preferable when the final log is too large for local storage, or if the logs are generated by and must be collected from multiple machines.

Synoptic desktop application. One drawback of a web service is the latency associated with uploading large logs — the user may want a quick analysis of a log that is local. Users with proprietary logs likely also cannot use a public web service. For these reasons, Synoptic is also available as a platform-independent desktop application. It can be used from a command line or from within a GUI (see screenshot in Figure 6). However, the Synoptic's web service includes more features than the desktop version.

Next, we detail how a user interacts with Synoptic.

4. Synoptic workflow

Synoptic's workflow targets effective support of log exploration through the lens of the generated model. The workflow aims to simplify the task of log parsing, so that users can spend most of their time analyzing the derived models. This section describes how logs are parsed, and some of the model analyses that Synoptic supports.

4.1 Extracting event instance triplets

Synoptic uses regular expressions [1] to extract event instance triplets from log lines the user considers relevant (see Section 2.1).

Division into executions. A single log file may record multiple executions of a system. Synoptic provides the user with two methods for breaking up a log into executions. The user may provide *delimiter* regular expressions, in which case an execution is a contiguous set of log lines between two matched delimiters. The user may also specify some number of *mapping* regular expressions, in which case fields parsed from the log line are mapped to some trace id. Lines mapping to a trace with the same id are interpreted as belonging to the same execution. For example, the traces in Figure 3 use the `Succeeded authorizing...` line as a delimiter.

Event abstraction. Users provide regular expressions to match log lines of interest. These regular expressions must all define a capturing group named `TYPE`, which indicates the abstract event-type that will be associated with the log line. This group's value may be a hard-coded string, a concatenation of multiple groups parsed from the log line, or a combination of the two.

Ordering events in an execution. To mine temporal event invariants from executions, events in an execution must be totally ordered. Users may specify the value to use for ordering as a built-in regular expression capturing group (e.g., `FTIME` for a float-based ordering), or use the order implicit in the log file line numbers.

4.2 Invariant selection

Synoptic mines three types of temporal invariants (Section 2) to guide refinement. The choice of invariants is important because they constrain the executions a derived model may generate. By default, Synoptic uses all of the mined invariants. However, users may know that some invariants are false because the logs do not sufficiently represent possible system behavior. Synoptic assumes that the user knows more about the system than is present in the log and allows users to mark some of the mined invariants as false so that Synoptic does not use them to over-fit the model to the log.

4.3 Model exploration

Synoptic presents users with interactive models. Users can tweak and explore them in pursuit of goals ranging from a more complete understanding of their system to identifying the source of unexpected behavior. A number of features are available to aid users in manipulating the models for these purposes.

4.3.1 Matching abstract and concrete information

The user can select a partition in the model and view the log lines that correspond to this partition (Figure 6). This is useful when the user wants to unpack the partition and identify the set of events that were actually logged at this point. The user can use this information to browse to a specific line in the log file that contains the interesting event. This operation is a kind of drilling down, mapping abstract information in the model to concrete events in the log.

Synoptic models are generative — they may accept traces that are not present in the input log. A user may want to know if a trace accepted by the model was observed in the log or not. For example, to a user, a generated trace may resemble buggy behavior, and the user may want to know whether the behavior actually occurred (if so, the system contains a bug). If a generated trace is invalid, it indicates that the input log is incomplete. This may lead the user to expand the test suite to invalidate an overfitted temporal invariant, which both improves the test suite and allows Synoptic to exclude the invalid generated path from the model.

To help users distinguish these two kinds of traces, users may select multiple partitions and consider the set of concrete traces that pass through the selected partitions. Synoptic either lists all the observed traces that pass through these partitions or lets the user know that the selected sub-trace was not observed. For example, the user may select the left most *login attempt* node, along with *auth failed* and *authorized* nodes in the model in Figure 6, and find out that there is indeed an observed trace that passes through these nodes (Trace 2 in Figure 3). This more advanced capability proved to be of particular use to developers in practice. It helped them understand unexpected paths in the model by exposing the associated concrete traces from the input log.

4.3.2 Filtering rare and common behavior

Sometimes Synoptic-generated models are large and contain more information than is necessary. For example, a developer may be interested in a section of the model when seeking to pinpoint a rarely-occurring behavior. To support this use-case, Synoptic allows the developer to filter out high/low probability edges from her view of the model. More generally, the user may be interested in high/low probability traces admitted by the model. Synoptic lets the user select a start and end node, and specify the maximum/minimum path probability to use for hiding all paths between the two nodes that have a path probability outside of the desired range.

4.3.3 Comparing models

A common use-case for Synoptic models is to study how the models change with different log inputs. Log inputs may differ because of additional executions in the log, a change to the mined set of invariants, a modification to the codebase, or because of added or removed logging statements.

Synoptic displays two models side by side and highlights their differences. These may be topological (e.g., the node count is different), or statistical (e.g., the transition probability of certain edges may have increased/decreased). By studying model differences, developers can check whether system behavior is the same or different — either over different traces or different system settings.

5. Related work

Numerous log analysis tools exist; however, we know of no freely-available tool to extract finite state machine models from console logs. A popular tool choice in the enterprise is Splunk[2], which supports various analyses and understands many common log formats. Splunk's main advantage is the scalability of its analyses due to MapReduce [6]. Popular tools that are similar to Splunk are Sawmill[3] and AWStats[4]. Synoptic supports log exploration, which is a more general goal than what is targeted by tools that have a tighter focus, such as Sisyphus[5], which targets anomaly detection.

Due to space constraints, we only briefly summarize related work that deals with specification mining. We introduced the basic BisimH algorithm in a workshop paper [9], and provided a rigorous evaluation of the approach, as well as positioned it among related work, in [3]. Perracotta [10] mines and visualizes temporal properties of event traces, and has been used to study program evolution. Unlike Synoptic, Perracotta does not use the mined temporal properties to infer a model of the system. The kTail algorithm [4], used extensively in related work (e.g., [2]), 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. Lo et al. [8] augment the kTail algorithm with temporal properties mined from execution traces to guide state merging while ensuring that the final model satisfies temporal constraints. Synoptic produces similar high-precision models while leveraging refinement, as opposed to coarsening, to greatly increase the efficiency and scalability of the approach.

Acknowledgments

This work is supported by the National Science Foundation under grants CNS-0963754, CCF-1016701, and under grant #0937060 to the Computing Research Association for the CIFellows Project.

References

[1] Pattern (Java Platform SE 7 b141) `http://download.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html`. Accessed July 1, 2011.

[2] 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.

[3] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proc. of FSE*, 2011.

[4] 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.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *Proc. of CAV*, 2000.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

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

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

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

[10] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proc. of ICSE*, 2006.

[2]`http://www.splunk.com`
[3]`http://www.sawmill.net`
[4]`http://awstats.sourceforge.net`
[5]`http://www.cs.sandia.gov/~jrstear/sisyphus`