Synoptic : Unraveling Distributed Systems with Message-level Summarization and Inference Techniques

Slava Chernyak Josh Goodwin Sigurd Schneider Ivan Beschastnikh

University of Washington
Computer Science and Engineering

1 Introduction

Distributed systems are often difficult to debug and to understand. Some of this difficulty can be ascribed to the complex behavior of the nodes comprising the system. For example, nodes may act concurrently with one another and may participate in multiple protocols simultaneously. Another difficulty is the distributed nature of state. No node can observe all the system state, and due to asynchrony in the networking environment observations of remote state are stale unless protocols with significant overhead are used.

As a result, developers who design and implement distributed systems often gain confidence in their system by adhering to strict software engineering principles and methodologies such as test-driven development, and by amassing sufficient experimental evidence (e.g. message and node state traces) to indicate that the system operates without faults. Both of these methods are insufficient for constructing robust complex distributed systems and are problematic for two reasons.

First, it is typically impossible to explore all states of a complex distributed system. Therefore model checking, and testing are only practical for checking correctness of a subset of system modules in a simple setting. Second, it is challenging to understand the behavior of a distributed system from collected traces without performing some form of aggregation and reasoning over the traces. As a result, the system developer is inundated with large node traces and has to manually parse these traces for signs of discrepancies and hints of system correctness.

We approach the problem of understanding system behavior as a summarization and a data reduction challenge. In this paper we describe the design and implementation of *Synoptic* – a tool to generate an Extended Finite State Machine (EFSM) representation of a distributed system execution by analyzing the network-level traces generated by nodes in a distributed system. Synoptic describes the behavior of a distributed system and reveals temporal and structural constraints relevant to this domain, such as those present in the payload of messages exchanged by nodes and in the ordering of messages in the traces. Although Synoptic reasons over the message traces, our representation of the distributed system is not a proof. We do not aim to generate system specification, instead our goal is to generate a *useful* summary representation of the observed system behavior. Additionally, we distinguish our problem from compression, which must retain *all* information. Our representation, on the other hand, can trade-off information content for conciseness. Additionally, unlike compressed content, our representation is intended to be inspected by a human. For this reason, the *form* of the representation is an important consideration.

To reach our objective we develop two algorithms. The first algorithm – the *GK-Tail* algorithm – is based on a paper by Lorenzoli, et. al. [9] in which program behavior is modeled with an EFSM. Their model captures information regarding both the temporal state graph structure of the program as well as the data constraints for each transition edge in this graph. To generate this EFSM, multiple traces are used where each trace consists of a sequence of method invocations annotated with values for parameters and variables. The resulting EFSM is beneficial in understanding program behavior and in generating tests with thorough coverage. We adapt this algorithm to our domain using the intuition that messages between nodes may be considered as remote method invocations.

Our second approach to the problem is a bisimulation-inspired algorithm we call *Bikon* [1]. This algorithm is novel because of a key property – the algorithm creates EFSMs that preserve all temporal properties of messages – e.g. message orderings. In addition, we use Daikon [5] to additionally guide the bisimulation towards representations that group together structurally interesting mes-

[1]Bikon = [Bi]simulation + Dai[kon]

sages. In particular, we apply Daikon when temporal properties are no longer sufficient to discriminate bisimulation states.

The contributions of Synoptic are the following. We develop and compare two algorithms to solve our task. We then apply both algorithms to real distributed systems to determine their performance and to compare the resulting representation for usability and information content.

We start the rest of the paper with a motivation for our work. We describe why we developed Synoptic and give a few example systems where it may be applied to great effect. We then overview our assumptions and design in Sections 4 and 3 in which we detail the parts of the system that collect and process system message traces. In Section 5 and 6 we elaborate our two algorithms – the adapted GK-Tail and Bikon. We finish the paper with an overview of our proposed evaluation in Section 7, our initial results in Section 8, and our related work in Section 9.

2 Motivation

To motivate the problem more concretely we present two example systems to which Synoptic might be applied to infer useful temporal and structural (e.g. data-oriented) properties. In our first example we describe an uptime monitoring applications that maintains the status of a set of nodes in the system. Our second example considers a MapReduce [3] system.

2.1 An Uptime Monitoring System

Consider an uptime monitoring system in which $node_1$ periodically sends ping messages to determine whether the other nodes in the system are still up. When the ping message is received by $node_i$, the node immediately replies with a pong message to $node_1$. When $node_1$ receives a pong from $node_i$, it sends a status query to $node_i$. This is the only pattern of communication in the protocol, and for now we assume that ping/pong/status interactions are not interleaved with one another. The captured trace of communication is a finite, totally ordered sequence of the messages sent and received by $node_1$.

Considering only message type equivalence, a possible summary of the observed behavior is that $node_1$ keeps sending ping messages, and if it receives a pong as response, it may send a status message. We would like to infer the following *temporal* properties to assure ourselves that the system is functioning correctly, at least in so far as we can observe it through the generated message traces:

- Pong is only received after sending ping.
- Status it only sent after receiving a pong.
- The transitive combination of the previous two, i.e. a status message is only sent after a ping has been sent.
- A ping may follow a ping.
- A pong need not be followed by a status message.

2.2 A MapReduce System

Our second example is that of MapReduce [3] – a distributed parallel data processing algorithm. It may be desirable to use Synoptic to examine a MapReduce system to verify its correctness, or to generate proper test cases. We first provide the necessary background on how MapReduce works, and then consider the kinds of properties that Synoptic might derive from a MapReduce implementation.

Background

The system consists of a set W of worker nodes, and a master node M (for simplicity we assume that there is just one master node). The basic flow of node interactions in the system is as follows:

1. M will send a "start map" control messages to some subset $S \subseteq W$ of the worker nodes. This should begin the "map" phase.
2. Each node $n_i \in S$ may send 0 or more messages to any nodes in W of the form $< k, v >$ corresponding to key-value pairs emitted during the map phase. Call the set of nodes that n_i sent messages to R_i.
3. Each node $n \in S$ will send a "map finished" control message to M to indicate it is done with the "map" phase.
4. After all "map finished" control messages have been received from all nodes in S, M will send a "start reduce" control message to a subset $R \subseteq W$ of worker nodes, exactly equal to the $R = \bigcup_{n_i \in S} R_i$, the set of nodes that were sent key-value pairs during the map phase.
5. Finally, each node $m_i \in R$, will send a "reduce finished" control message to M. After all these messages have been received from all nodes in R, the job is complete.

System Properties

The first goal of Synoptic for this system is to model the phases of the system as an EFSM. Examples of useful features of such an EFSM representation are the following:

1. An EFSM generated for each worker node should represent a local ”map” and ”reduce” phase signaled by the sent and received control messages. Specifically this means inferring that no ”start reduce” control message was received by the node during the ”map” phase, no key-value messages were sent during the ”reduce” phase, and all proper control messages were sent and received.

2. An EFSM generated for the master node should represent a global ”map” and ”reduce” phase. The end of the map phase should be signaled by receiving ”map finished” control messages from all nodes in S, similarly the end of the reduce phase should be signaled by receiving ”reduce finished” control messages from all nodes in R.

3. A cross-product of the local EFSMS generated for every node might reveal that the set R of nodes that were sent ”start reduce” control messages from M is the same as the set sent key-value messages during the ”map” phase.

Additionally, when we bring the contents of messages into the mix, we may want to infer further properties. For example:

The keys sent as part of the key-value messages from mappers to reducers satisfy the following constraint: The keyspace $\mathcal{K}$ is partitioned into disjoint subsets $K_1, K_2, \ldots K_j$ by some partition function, where $j = |R|$ the number of reducers. The partition function is set, and fixed, for the duration of the job and may be arbitrary. There exists a map between each K_i and $n_i \in R$, call it g, such that $g(k) = n_i$ whenever $k \in K_i$, that is g assigns the message to a reducer based on key. When a key-value message $< k, v >$ is sent from some mapper node, it is sent to the reducer assigned by $g(k)$.

Ideally this may be inferred by Synoptic and would be useful for verification or test case generation. This would make sure, for example, that a message $< k, v >$ is never sent to some node n such that $g(k) \neq n$.

3 Assumptions

We make a few simplifying assumptions concerning the environment or the distributed system we analyze.

- We assume a non-broadcast networking medium at the message capture layer. For example, we do not handle wireless environments in which the nodes in the system sniff and thereby receive all messages. However, we can operate in a wireless environment with all nodes restricted to using TCP for communication.

- We assume that the nodes are using a reliable communication protocol so that messages are never lost. However, we do allow reordering of messages to occur.

- We assume that the set of nodes in the system is constant and does not change throughout the trace. That is, nodes do not fail and new nodes do not join the system.

- If the system is composed of just one physical host the local clock can be used for timestamping all messages. To analyze the behavior of multiple network nodes, a consistent notion of time is necessary. To start with, we assume correctly synchronized clocks (e.g. nodes use NTP). Later, we hope to establish a global partial ordering on the messages in the system with vector clock algorithms [8].

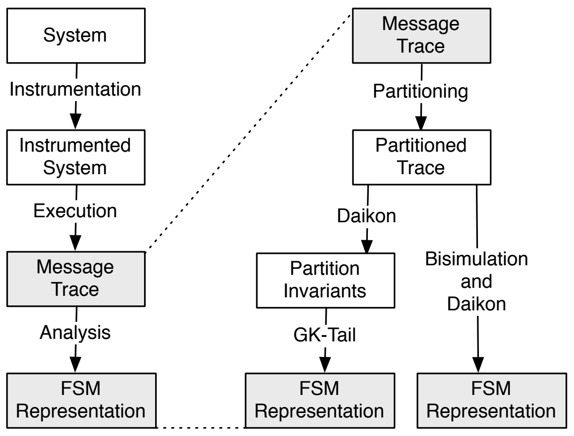

Figure 1: An overview of the system pipeline (left) and our two analysis pipelines implementing our analysis algorithms (right).

4 System Design

Synoptic is comprised of several components. Given an unmodified system as an input, we instrument the system to capture all network-level messages into a trace. For each message in this trace, we additionally associate useful meta information such as the source, destination node addresses, and a timestamp of the message. Because we assume that messages are never lost (see Section 3), we only capture outgoing messages. The traces are then collected from the all the nodes, and are then merged into a single trace. We assume that the nodes use NTP or a similar protocol to synchronize their local time.

Next, Synoptic infers properties and relationships between messages in the trace that lead to a more concise representation. As part of this step, the trace may be partitioned and different representations may be generated for different partitions of the trace. Moreover, information content may be elided and excluded to make the representation more concise. Finally, the derived representation is presented to the user, along with all instances where information loss might have occurred. Figure 1 illustrates the components in the system's pipeline and the two types of analysis supported by Synoptic. In the rest of this section we elucidate those parts of our system that do not involve the analysis step.

4.1 Message Capture

We use the term *message* in a general sense to refer to a formatted message sent between two nodes in a distributed system. We assume that all messages have a source address, a destination address, a timestamp, and a data payload that may or may not be used in the analysis.

There are several ways to capture messages. For example, all traffic on a local machine could be sniffed (e.g. using `tcpdump`), or the system could be modified to use a custom network library that logs all messages sent and received to persistent storage. Initially we will explicitly capture messages by modifying the distributed systems that we study.

To perform analysis on message payload, we will use protobuf [1] specifications of protocol messages exchanged between nodes in the system. This specification captures all the message fields and their types. The user will annotate the protobuf specifications of the system to indicate which fields should be used by Synoptic as part of its analysis. Figure 2 gives an example protobuf specification for messages that represent an address-book record describing a person with a name, email, and a number of phone numbers.

The availability of a protobuf message format specification enables semantic analysis of the messages observed. They provide a natural way to distinguish messages according to their type, and help inferring relations between the message payloads. Consider, for example, a address book server that accepts messages carrying an email as payload, and responds with a person message (see Figure 2) that contains the phone numbers the server knows about. Structural information about the payload of the messages makes it easier to infer, for example, the invariant that the email field in the request message is identical to the email field in the person message sent as a response.

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2
 }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

Figure 2: Example protobuf format representing a person type that has a name, email, and a number of phone numbers; and an address-book type that encapsulates some number of person records.

5 The GK-Tail Algorithm

The paper by Lorenzoli, et. al. [9] presents a method for generating EFSMs from method call traces where the states of the EFSM represent approximate program states, and the edges are annotated with relevant constraints on the data associated with method calls. Their pipeline is as follows:

1. Merge input-equivalent traces. This is an initial data-reduction step. Input-equivalent traces are those that represent a sequence of the same method calls but with potentially different parameters. The annotations on the edges between states now enumerate the set of all observed parameters for the merged arcs.

2. Use `Daikon` [5] to come up with the most general representations for the sets of values on the arcs. This produces an initial EFSM.

3. Use the *GK-Tail* Algorithm to reduce the number of states in the EFSM by merging state sequences which have similar *k-futures* based on either equivalence or subsumption similarity metrics.

In our adaptation of this inference approach, we will map the algorithms from [9] into the distributed systems domain. The intuition we will exploit is that a trace of method calls made by a program is fundamentally similar to the trace of messages sent and received by a node.

5.1 Performing Inference

More formally, we propose the following modifications to be prepended to the pipeline described above:

1. The input is the global trace consisting of all of the events in the distributed system.
2. We partition it into a set of traces using a partition function. The choice of partition function is part of our investigation. One obvious candidate is a partition function that splits the trace based on host. Thus it produces a trace per host.
3. For each such trace we define a map that will map the trace of messages into a trace of method calls. The correct choice of such a mapping is also part of our investigation.

An intuitive candidate for such a map is as follows:

- Let the ”type” of a message refer to the message signature defined by all the fields in the message. Let the ”src” and ”dest” of a message refer to the source and destination. Then we map the tuple ¡type,src,dest¿ to a unique method name. In other words, messages of the same type sent from between the same source and destination will be represented as a particular method call in the trace.
- We augment the fields of the message a timestamp, and potentially a latency value. We map from this augmented set of fields to method call parameters.
- We map every instance of incoming and outgoing message to an instance of a corresponding method call in the trace with corresponding parameter values, as defined above.

Thus we map the partitioned trace from a distributed systems domain to a a set of method call traces. We then apply the pipeline described above to the trace.

5.2 Future Work

We hope that the results from above will direct our further work. Specifically here are the possible directions that we may pursue next:

- We may find that it is necessary to refine the approach of the *GK-Tail* algorithm to better suit our domain.
- The resulting model may be a useful concise representation of the data, and may allow us to mine/infer temporal patterns. The grammar of such patterns will be part of our investigation. As a starting point, we intend to base our patterns on the specification patterns and classification scheme for these patterns formulated by Dwyer et. al. in [4].
- It may be interesting to investigate the difference between a product of the local models generated from traces at each node, versus one that is generated from a global trace. The analogy of this is given a brief mention by Lorenzoli, et. al. Specifically, they consider a multi-threaded environment and conclude that their approach may be applied to each thread individually, or to a global trace with interleavings as they occur during execution. This is a clear analog to the distributed systems case with one important exception - simultaneous events are now possible, and arbitrarily ordering them may introduce bias into the inference. It might therefore be necessary to extend the representation to capture concurrency.
- It may be possible to further refine the model by statistical methods. Specifically, it might be necessary to ignore parts of the trace to come up with a concise and representative model. This may correspond well to accounting for failure or other unexpected events in the distributed system. The specific statistical models or machine learning approaches that may be necessary to do this might be part of our investigation. This will be approached as necessary – when we scale to larger and more complex distributed systems we may find empirically that we cannot generate concise and useful models if we include all of the data in the analysis.

6 The Bikon Algorithm

Bisimulations are simulation relations that provide a strong notion of similarity for relational structures. They emerged in different fields [11], and their key property is preserving certain properties of the relational structure, for example, two strongly bisimilar transition systems are guaranteed to satisfy the same set of LTL formulas. Building on this property, an important applications in model checking is model minimization [6]. We modified a partition refinement algorithm [10] to build a bisimulation-inspired minimization that we call *Bikon*.

The basic idea is to interpret observed messages as states S in a transition system $(S, A, \{\xrightarrow{a} \mid a \in A\})$, where the last component is a set of relations on S for every labels from the set A. Edges can be inserted to capture a notion of time, a notion of request/response extracted from source/destination fields, etc. At this point, the graph is merely a different representation of the trace. Running a bisimulation minimization algorithm on this will group together messages if the corresponding states satisfy the same set of modal formulas.

6.1 Relaxing bisimulation

As it turns out, bisimulation is a too strong notion of similarity, since it cannot abstract from noise, seldom error conditions, package loss, and most importantly, will never collapse finite acyclic traces into cyclic ones. We thus modify the stratification based algorithm for bisimulation to the following:

Definition. *Let* $(S, A, \rightarrow)$ *be a transition system and* $\rho \in \mathbb{R}$. *Let*

- $S/_{\approx_0} = S \times S$
- $S/_{\approx_{n+1}} = S/_{\approx_n} \cup \{\Delta', \Delta''\} - \{\Delta\}$ *such that there is* $a \in A$ *with*

  (a) $\Delta' = \{s \in \Delta \mid \exists s' : \ s \xrightarrow{a} s'\}$

  (b) $\Delta'' = \{s \in \Delta \mid \forall s' : \ s \overset{a}{\not\rightarrow} s'\}$

  (c) $\dfrac{|\Delta'|}{|\Delta|} \in [\frac{1}{2} - \rho, \frac{1}{2} + \rho)$

Then $\overset{\rho}{\approx} = \bigcap_{n \geq 0} \approx_n$ *is the* ρ*-condensation of* $(S, A, \rightarrow)$.

The definition above differs from strong bisimulation in condition (c). Condition (c) allows splits only if the resulting partitions will be evenly sized. The underlying assumption is, that noise, seldomly occurring error conditions, rarely used protocol features, etc, do comprise the majority of observed behaviors in a trace. Thus, it will not cause a split. On the other hand, frequently observed behavior will most probably be related to a protocol feature, and a split is allowed. From a information theoretic perspective, this splitting heuristic avoids over-fitting the observed program traces in that it does not model rarely occurring behavior. It is possible to begin with a finer relation than $S \times S$, and the algorithm will never return a coarser relation. Note that for $\rho = \frac{1}{2}$ the algorithm coincides with strong bisimulation minimization, whereas for $\rho = 0$ no partition is ever split, so the result depicts relations between possible initial partitioning. The results obtained for greater values of ρ correspond to less detail; interestingly this does not mean that smaller values of ρ imply a greater level of information, since too much reduction is unproductive.

6.2 Using structural properties for arbitration

The above algorithm captures temporal properties of a message trace. However, it does not in any way consider the structural properties of the messages. We use Daikon to arbitrate those cases where the relaxed bisimulation algorithm is unable to decide whether or not to split a node. In particular we use Daikon for those nodes that have reached equilibrium with respect to ρ.

We consider the set of invariants that are associated with messages corresponding to two edges – the edge that induces the split (i.e. which is present in only one of the partitions), and the same edge in the unpartitioned node. We consider the sub-trace of messages corresponding to the two types of messages (e.g. source and destination of each edge) and employ Daikon to generate a set of invariants for the two edges. We then compare the set of invariants and use a metric to decide which edge has more structural properties – this is the edge we keep. As one metric, we can use the raw number of generated invariants.

6.3 Limitations

A problem with the algorithm above is that certain value of ρ can trigger many split, which means that the granularity of the output graph cannot be adjusted continuously. Another issue with the algorithm is its non-determinism. There are situations in which two splits are possible, and pursuing either of the splits will make the other impossible afterwards. Both issues might indicate that there needs to be a more sophisticated mechanism to decide when to split partitions.

7 Proposed Evaluation

To evaluate our work we will run the program on selected distributed systems, including Hadoop [12] and Harmony [2]. The evaluation criteria of the representation output by Synoptic properties will include conciseness information content, and usefulness.

To evaluate the system in [9], the authors use a test coverage metric. By ensuring that the set of tests covers every transition arc in the EFSM representation, the tests can be considered to have a high degree of coverage. In our work, we plan to see if more thorough test cases can be generated by analyzing our resulting EFSM representation than would be obtained by other invariant and FSM methods.

7.1 Small and Interesting Distributed Systems

We plan to drive the development of Synoptic by applying it to interesting networked test program. Here is a set of such programs that we think are appropriate for this purpose:

- A `ping-pong` system which consist of two nodes each oscillating between two states. It will be the job of our tool to infer these state sequences.
- A `demux` node, which forwards incoming packets to different nodes according to properties of the payload. We want to infer the relation between payload and forwarding destination.
- A storage and retrieval system. The goal of our tool here may be to infer the constraint of a retrieve request always returning a modified result after a store request.
- Other ideas include more complicated client-server systems where concurrent client access may force the inference engine to consider how to represent synchronization and concurrency.

7.2 Evaluation Criteria

As part of evaluation we will use people to evaluate our generated EFSMs. For instance, we will ask a person to consider the raw trace and to describe what the system does. We would then compare their description with what another person would describe when considering the generated EFSMs. We believe that the EFSMs would make it much easier to investigate system behavior and that it would greatly reduce the amount of time a person needs to spend to understand the various protocols inside a distributed system.

8 Preliminary Results

8.1 GK-Tail Algorithm

We have created an incomplete version of the *GK-Tail* algorithm informed by [9]. This algorithm will consider *k-Futures* of states, which can be loosely defined as the set of all sequences of length k, and will merge state sequences whenever their *k-Futures* are equivalent. The algorithm iteratively considers pairs of states to be merged.

The current implementation is incomplete, since the complete *GK-Tail* algorithm allows for merging of states that have *k-Futures* that are not just equivalent, but satisfy a strong or weak subsumption relation.

With the incomplete algorithm we were able to transform the degenerate EFSM representing the messages

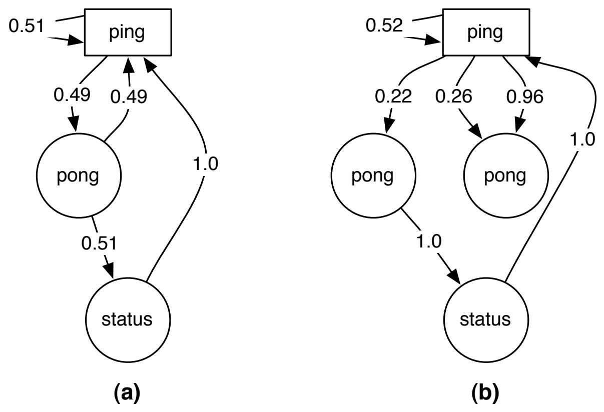

Figure 3: (a) A step in the Bikon algorithm in analyzing the ping-pong system (b) The next step in the Bikon algorithm that partitions the pong node into two nodes, one of which always transitions to a status node.

sent in a ping/pong system, into the most concise representation. The degenerate EFSM in this case refers to the EFSM consisting of a state and transition for every ping and pong message sent. Thus if n ping/pong messages were present in the trace, the degenerate EFSM will contain $n + 1$ states with each state having exactly one successor, and the arcs representing an alternation of ping and pong messages. The most concise representation is an EFSM with exactly two states.

8.2 Bikon Algorithm

We tested a preliminary version of the Bikon algorithm on an artificially generated trace of the uptime monitor program described in Section **??**. For the generated trace we set the probability that a ping was responded with a pong, and that a ping was responded with a status, both with probability of .5. The analyzed trace was 100 messages long.

We were able to obtain the transition systems depicted in Figure provided that we did not allow merges of messages of different types. This means that result (a) is the most condensed graph. As a next iteration of the Bikon Graph, (b) has two different nodes representing pong messages. This is because roughly half of the pong messages exhibits a different behavior than the other messages in that they are followed by a status, and the others do not.

The deeper insight from two different nodes with the same label seems to be that some property of the state that is split causes different behavior afterwards. Daikon could be used to extract invariants of the payloads of the messages in both states, which might reveal that status is only sent if the number in the payload exceeds a certain range, for example. This is our next step in developing

Bikon.

9 Related Work

Daikon [5] is a tool to detect likely program invariants from observed program executions. During analysis of the execution traces, at marked program points Daikon instantiates a set of template invariants and tracks those invariants that hold true for the observed sequence of variable values. We plan to employ this technique to relate data fields of messages, and to find data invariants that hold between messages. Dynamically inferred program invariants have also been used by DIDUCE [7] to report invariant violations that might indicate software bugs. Unlike Daikon, DIDUCE is an online tool that can be run to learn program invariants, as well as to check previously learned invariants. To infer invariants, DIDUCE instruments Java programs to output value of tracked expressions and uses a relaxation method by which strict invariants that match initial values are relaxed over time to accommodate new information.

Distributed systems are notoriously difficult to get right. Recent efforts by the systems community target bug finding in distributed systems with model checking. MODIST [14] explores the interleavings of concurrent events in an unmodified distributed system, thereby model checking the live system. CrystallBall [13] explores the state space of an actively executing distributed system, and when inconsistencies in possible future states are found, CrystallBall can be used to steer the distributed system away from buggy states. These tools, however, do not target the extraction of system properties that may be used to understand the behavior of the system they check. Property representations produced by Synoptic may be leveraged by these tools to guide model checking, and Synoptic may use these tools to target system execution towards states for which Synoptic lacks information to derive a concise representation.

Perracotta [16] is a tool to mine interesting temporal properties from program event traces. It has been used to study traces of function calls for program evolution [15]. It has also been made scalable and robust to analysis of imprecise event traces to understand behavior and uncover bugs in very large codebases such as the Windows Kernel and the JBoss application server [17]. Perracotta first instruments the program to output event information – all prior work uses method entrance and exit points. Second, the instrumented program is run – Perracotta relies on extensive test suites and random exploration of method calls to generate a broad range of event traces. Finally, the generated traces are used to infer program properties. The system generates candidate temporal patterns and then attempts to gain evidence that indicate the pattern holds by scanning through the trace. Patterns are expressed in terms of quantified regular expressions which are similar to regular expressions. The system uses a partial order hierarchy of properties that are built on the response pattern – the simple cause effect relationship between some two events.

Unlike Synoptic, Perracotta considers a totally ordered trace of events and does not consider properties that might be of interest in the domain of distributed systems. For instance, events may be concurrent and cannot be ordered with respect to one another. Additionally, Perracotta does not make use of any relationships between event data (e.g. method arguments/message payload).

The work by Lorenzoli, et. al. [9] is most closely related to ours, and inspired our methodology. In this work, multiple interaction traces are used to automatically generate an Extended Finite State Machine (EFSM). Each trace consists of a sequence of method invocations annotated with values for parameters and variables, and is generated by a single execution run of the program being analyzed. The resulting transition arcs of the EFSM are annotated with the relevant constraints on the data to transition from the start state to the next state, providing a behavioral view of the program that includes how the input data affects the behavior. With Synoptic we extend the work by Lorenzoli, et. al. to the domain of distributed systems.

References

[1] Protocol Buffers - Google's data interchange format, phttp://code.google.com/p/protobuf/. Accessed January 13, 2010.

[2] Harmony: A Consistent Distributed Hash Table, http://harmony.cs.washington.edu/. Accessed January 13, 2010.

[3] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[4] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, New York, NY, USA, 1999. ACM.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 213–224, New York, NY, USA, 1999. ACM.

[6] K. Fisler and M. Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39–78, 2002.

[7] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM.

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

[9] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 501–510, New York, NY, USA, 2008. ACM.

[10] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

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

[12] W. to Apache Hadoop! http://hadoop.apache.org/. Accessed January 13, 2010.

[13] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Crystalball: predicting and preventing inconsistencies in deployed distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.

[14] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: transparent model checking of unmodified distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[15] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 340–351, Washington, DC, USA, 2004. IEEE Computer Society.

[16] J. Yang and D. Evans. Dynamically inferring temporal properties. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 23–28, New York, NY, USA, 2004. ACM.

[17] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM.