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.

1.1 Project Description

In the paper by Lorenzoli, et. al. [8], program behavior is modeled by an Extended Finite State Machine (EFSM). This 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 can be beneficial in understanding program behavior and in generating tests with thorough coverage.

In our project we will design and implement *Synoptic* – a tool to generate an EFSM representation of a distributed system execution by analyzing the network-level information generated by nodes in a distributed system. In addition, *Synoptic* will describe the behavior of a distributed system and reveal data constraints relevant to this domain, such as between messages exchanged by nodes.

Usually the system developer is inundated with information about what each node in the system does. We approach the problem of understanding system behavior as a summarization and a data reduction challenge. We aim to mine patterns and relationships that range over the temporal properties of the captured system trace as well as the trace data itself. The resulting EFSM representation will need to have a usable balance between two contradictory goals:

- The representation should be as concise as possible
- The representation should retain as much information as possible

We foresee the research contribution of our project to be a characterization of a sweet spot between these two goals in the domain of representing executions of distributed systems. We also aim to define the metrics of conciseness and information in a way that reflects sound judgment of distributed system designers.

Note that unlike the problem of compression, which must retain *all* information, our representation 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.

Finally, although we employ inference and reason over the message trace, our representation of the distributed

system is not a proof. We do not aim to generate system descriptions that hold in all circumstances. Instead we aim to generate a *useful* summary representation of the observed system behavior.

2 Motivation

To motivate the problem more concretely we present two example systems to which Synoptic might be applied to infer useful temporal and 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 is more involved and 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 properties:

- 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.

We can create an finite state automaton with these properties from the observed traces by taking as states the types of messages, and insert transitions between states s_i and s_j whenever there are messages of those types such that s_i is a direct successor of s_j. Figure 1 shows a sample diagram, with relative frequencies of successor messages. Note that these do not necessarily add up to one, as we abstract infrequent messages away.

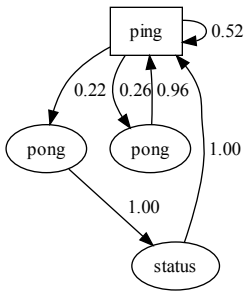

Figure 1: A sample automata representing the communication pattern in the example uptime monitoring system.

Let us now take into account the messages source and destination fields. We want to identify ping/pong/status interactions, i.e. infer that a status to $node_i$ only occurs if a ping message was previously sent to $node_i$. We need to infer a relation between messages that have identical, but swapped source and destination fields. Note that based on source and destination fields we can also interpret interleaved traces resulting from concurrent communication with several nodes.

Finally, consider a modified service discovery program in which every pong message carries a data field containing the uptime of the node, and in which status messages are only sent if this number is low, i.e. indicates that the node recently rebooted. The first problem here is that a simple message comparison to find redundancy will conclude that most pong messages are in fact unique. To obtain the results from the above, the data field can be explicitly ignored, but a more sophisticated approach is desirable. The system must partition the pong messages according to their behavior, i.e one partition for pong messages that are followed by a status message, and one for those that are not. We then can use a tool like Daikon to reveal that the data field values of the messages in one partition are below a certain threshold. In this way we can infer that status messages existence is strongly correlated on a property of a message data field.

2.2 A MapReduce System

Our second example is that of MapReduce [3], which has been extensively used for distributed parallel data processing. 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. First, 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 receive all messages. However, we can operate in a wireless environment with all nodes restricted to using TCP for communication. Second, 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.

4 Overview of Approach

Synoptic is comprised of several components. First, all messages sent and received by the nodes of a distributed system are captured. This data is then pre-processed, i.e. reduced to the features of interest and content with 0 information is removed. 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.

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, 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.

4.2 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 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.

4.3 Capturing the notion of Time

If the system is composed of just one physical host the local clock can be used for timestamping all messages.

```
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.

To analyze the behavior of multiple network nodes, a consistent notion of time is necessary. To start with, we will assume correctly synchronized clocks (e.g. using NTP). Later, we hope to establish a global partial ordering on the messages in the system with vector clock algorithms [7].

4.4 Performing Inference

The paper by Lorenzoli, et. al. [8] presents a method for generating Extended FSMs 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. To do this the system first performs an initial data reduction and pruning to reduce the state space. Then the system uses *Daikon* [5] to generate predicates associated with traces. An initial EFSM is thus created. It is further reduced by merging states that have equivalent "tails" of subsequent states. To do this last step they introduce the *GK-Tail* algorithm.

Our initial inference approach will involve mapping the algorithms from [8] 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. More formally, we propose the following mapping for

the local trace of each node in a distributed system:

- We map from the "type" of a message (where "type" may refer to the signature defined by all the fields in the message) to a unique method name.
- We augment the fields of the message with a source, destination, and timestamp, and 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 a local trace from a distributed systems domain to a method call trace. We then apply the algorithms described by Lorenzoli et. al to the trace. We hope that the result will direct our further work. Specifically here are the possible directions that we may then pursue:

- 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 when merged with other local models generated from other nodes in the distributed system 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 analogue 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.

5 Evaluation

To evaluate our work we will run the program on selected distributed systems, including Hadoop [9] 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 [8], 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.

6 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 [6] 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 [11] explores the interleavings of concurrent events in an unmodified distributed system, thereby model checking the live system. CrystallBall [10] 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 [13] 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 [12]. 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 [14]. 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. [8] 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.

7 Timeline

- **January 25** : Related work review done
- **January 28** : Algorithm finalized
- **January 30** : Evaluation plan done – includes test programs, metrics, and reasoning behind selection
- **February 3** : Initial version of report due [intro, algorithm details, related work, evaluation plan]
- **February 6** : Test programs done
- **February 16** : Hadoop and Harmony instrumentation done
- **February 24** : Report with partial results due [running end-end prototype implementation that can generate some results]
- **March 10** : Experimentation complete
- **March 12** : Final report due
- **March 15** : (week of) Presentation

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] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.