Inferring Networked System Models from Behavior Traces

Ivan Beschastnikh
Computer Science & Engineering
University of Washington
ivan@cs.washington.edu

ABSTRACT

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

To provide developers with more insight into networked systems I am working Dynoptic, a tool that infers a concise and accurate system model, in the form of a communicating finite state machine [5] from logs. Developers can use the inferred models to understand behavior, detect anomalies, verify known bugs, diagnose new bugs, and increase their confidence in the correctness of their implementation. Unlike most related work, Dynoptic does not require developer-written scenarios, specifications, negative execution examples, or other complex input. Dynoptic processes the logs most systems already produce and requires developers only to specify a set of regular expressions for parsing the logs.

Categories and Subject Descriptors: D.2.5 [Testing and Debugging]: Distributed debugging
Keywords: Dynoptic, CFSM, Modeling

1. INTRODUCTION

Understanding a system's behavior is a difficult development task that is required when a system behaves in an unexpected manner or when a developer must make changes to code they did not write. Logging and log analysis of captured system behavior is one of the most effective tools for building understanding. Unfortunately, the size and complexity of logs often exceed a developer's ability to make sense of the captured data. For example, production systems at Google log *billions* of events each day; these are stored for weeks to help diagnose errant future behaviors [15].

One promising approach to help users make sense of complex executions is model inference. The goal of a model inference algorithm is to produce a model, typically a finite state machine, that accurately and concisely represents the system that produced the input log of executions. Numerous such algorithms and corresponding tools already exist to help debug, verify, and validate systems [4, 11, 12, 13].

Unfortunately, it is challenging to apply this rich body of work to *networked* systems. This is because a common assumption made in model inference is that the underlying set of executions is totally ordered — for every pair of events in an execution, one precedes the other. This is not the case for logs of networked and distributed systems, where events at different nodes may occur without any happens-before relationship [10]. Therefore most model inference algorithms output finite state machine (FSM) models, which are difficult or impossible to map back to the multi-process implementation that generated the log.

The goal of my dissertation work is to develop model inference techniques that can be applied to logs of networked systems, such as protocol message traces. These techniques are being implemented as part of a tool called Dynoptic.

2. DYNOPTIC OVERVIEW

Dynoptic takes a log of observations as input and outputs a model that describes the system that generated the observations. Dynoptic is intended to model communication protocols and networked systems. I have experimented with a variety of formalisms (e.g., petri nets). However, an appropriate formalisms must be familiar to developers, intuitive, and simple to learn. The communicating finite state machines [5] (CFSM) model is similar to the traditional FSM formalism, and fits these requirements.

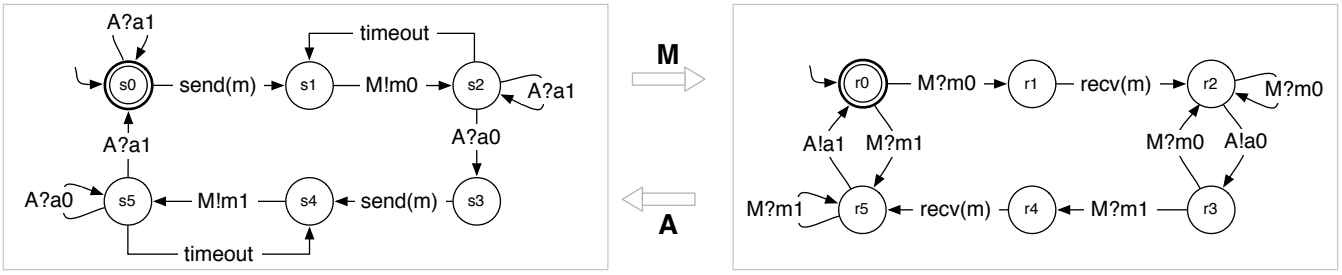
In a CFSM model, each process is specified as a finite state machine (FSM), and processes communicate with one another by exchanging messages over FIFO queues. Figure 1 shows a CFSM model for the alternating bit protocol.

Dynoptic expects a log of process-local events, as well as message send/receive events. It also expects each event in the log to be timestamped with a vector clock value, which is used to track the partial order of events in the system [10].

(1) First, Dynoptic parses the log and mines a set of *temporal invariants* that relate the logged events [3]. These invariants are temporal. One example is: "e1 is always concurrent with e2", meaning there is never a happens-before [10] relationship between the two events in any of the traces. Dynoptic guarantees that the mined invariants will be satisfied by the final model that it outputs to the user.

(2) Next, Dynoptic builds a graph of all possible serializations of events in the log. This graph is used to capture the possible interleavings of events in the DAG formed by the partially ordered vector clocks recorded in the log.

CoNEXT Student'12, December 10, 2012, Nice, France.



(a) Sending process **(b) Receiving process**

Figure 1: A CFSM model of the alternating bit protocol with (a) sending process model, and (b) receiving process model. The sender transmits messages to the receiver using queue `M`, and the receiver sends acknowledgments to the sender using queue `A`. Event `Q!x` means enqueue message `x` into queue `Q`, and event `Q?x` means dequeue message `x` from the head of the queue `Q`.

(3) The serialization graph from step (2) is abstracted into a *partition graph*. A partition graph is a partitioning of nodes in a serialization graph that conserves edges.

(4) The abstract serialization graph captures all of the observed executions, although it might be an over-generalization. At the core of Dynoptic is a counter-example based abstraction refinement loop [7]. Dynoptic uses the McScM model checker [9] to check the CFSM corresponding to the abstract serialization graph against each of the mined invariants. If an invariant is invalid in the abstract serialization graph, then the model is refined to satisfy the invariant — a partition is split to create less abstract model. Once all of the invariants are satisfied, Dynoptic returns the CFSM corresponding to the final abstract serialization graph.

3. RELATED WORK

Prior work on inferring CFSMs from observations is theoretical and focused on inferring models with desirable properties, such as models with no deadlocks and no unspecified receptions [6]. Such properties are important in theory, but we have found them to be less essential to our use-case — to generate models that provide developers with insight into a networked system implementation.

The most closely related non-CFSM model inference work to Dynoptic is our previous work on Synoptic [3], which is a tool for inferring FSM models from a log of sequential executions. The approach in Dynoptic parallels that of Synoptic, which mines temporal invariants, forms a small initial model, and then progressively refines this model to derive a final model that satisfies all of the mined invariants. The Dynoptic process, however, has to account for richer kinds of invariants [2], deals with a new model type, and is intended to provide insight into partially ordered logs.

Dynoptic relies on the McScM model checker [9, 8] for checking the validity of mined invariants in a CFSM model. McScM is one of the most advanced verification tools for concurrent systems; building on prior state of the art [1].

Most prior work on model inference assumes that the input log is totally ordered [4, 12, 13, 14]. This constrains the generated log to either exclude concurrency, or to capture a particular interleaving of concurrent events. Our position is that capturing concurrency as a partial order is useful and often indispensable for understanding a system's behavior.

Acknowledgements

Dynoptic is supported by Google, DARPA grant FA8750-12-2-0107, and NSF grants CNS-0963754 and CCF-1016701. I would also like to acknowledge the core Dynoptic project team: Michael D. Ernst, Yuriy Brun, Arvind Krishnamurthy, and Thomas E. Anderson.

4. REFERENCES

[1] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A Tool for Reachability Analysis of Complex Systems. In *CAV*, 2001.

[2] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. Mining Temporal Invariants from Partially Ordered Logs. *OSR*, 45(3):39–46, Dec. 2011.

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

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

[5] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *JACM*, 30(2):323–342, Apr. 1983.

[6] X. J. Chen and H. Ural. Construction of Deadlock-free Designs of Communication Protocols from Observations. *The Computer Journal*, 45(2):162–173, Jan. 2002.

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

[8] A. Heussner, T. Gall, and G. Sutre. Extrapolation-Based Path Invariants for Abstraction Refinement of Fifo Systems. In *SPIN*, 2009.

[9] A. Heussner, T. Le Gall, and G. Sutre. McScM: a general framework for the verification of communicating machines. In *TACAS*, 2012.

[10] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, July 1978.

[11] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *ESEC/FSE*, 2009.

[12] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, 2008.

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

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

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