Distributed Test Case Generation using Model Inference

Stewart Grant, Ivan Beschastnikh
University of British Columbia

1 Introduction

Developers of distributed systems strive for correctness by writing tests. Developing these tests is a difficult and error prone task which spans the life of a system. In spite of developer effort, complicated bugs frequently sneak through and are admitted into production systems.

State-of-the-art techniques for achieving correctness are widespread in their approaches and effectiveness. Verification aims to mathematically prove systems correct. Various specification languages leverage powerful model checkers to compute these proofs [7, 6]. Projects such as Verdi and IronFleet [8, 5] have demonstrated the utility of these techniques by developing fully verified systems. These proofs however require expertise, and effort.

Conversely, fuzz testing provides a low cost, low effort black box approach to improving system durability by automatically generating new test cases [10]. Fuzzers pointed at network endpoints allow developers to stress their software without writing tests. While efficient, fuzz testing only catches early bugs, and admits complex bugs.

In the middle are transparent model checking procedures such as MODIST [9]. Given a specification of correct behaviour (e.g. invariants), such tools exercise real systems, scheduling and interleaving distributed events using a centralized controller. Such techniques are useful for identifying and replicating complex distributed bugs. But, error states are reached using unguided brute force search! The time to detect bugs using such frameworks is long, and while detected bugs can be complex, they are often errors reachable shortly after initialization

Due to their low effort and high utility, tools such as MODIST are extremely powerful, but they can be largely improved. We posit that brute force search of bugs detection can be improved by extracting program specific information. Specifically that dynamic traces of program state can be used to model a systems behaviour, and guide transparent model checkers towards deeper bugs.

In this work we propose a novel technique for synthesizing models of distributed systems using dynamic state based analysis. Our modelling and checking algorithms are designed to analyze several GB's of logs generated

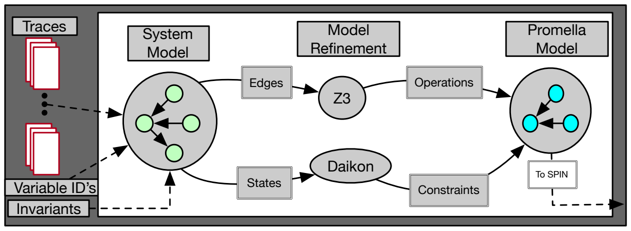

Figure 1: Dara pipeline: traces, variable IDs, and invariants are FSM model input. Z3, and Daikon refine, and add constraints. Refined models are translated to Promella for model checking with SPIN.

by real systems. Figure 1 overviews Dara's model generation pipeline. Dara builds state machines from logs, merges states to reduce complexity, and infers state properties to reduce false positives. Dara use off the shelf model checkers to exercises models and output *approximate* traces which are likely to, but may not correspond to real bugs. These approximate traces are then passed to a transparent model checker which validates, or invalidates potential bugs.

The Dara[1] tool is an implementation of our analysis tool-chain. The tool is written in Go, and analyzes on systems written in Go.

2 Challenges

Trace Collection Dara requires large amounts of program data to construct an accurate program model. Traces which capture diverse system behavior increase model accuracy. Aforementioned test suites, fuzzers, and transparent model checkers are each viable sources of program.

System Modelling We propose that an approximate system model can be extracted with a traditional test suite, and profiling, from observed state. Our approach merges system logs into a finite state machine (FSM). Each FSM state is defined by a unique combination of concrete variable values extracted from a log. State transitions (edges)

[1]http://www.github.com/wantonsolutions/dara

are defined by the sent or received messages, or important local events (syscalls, timers).

Model Accuracy A model from dynamic traces is approximate. Atomic steps in time must be defined only on the visible traces. Further, as a model becomes more precise (includes more variables) the state space of the model grows exponentially. Choosing variables to model is critical. The omission of a key variables produces inaccurate models. Such cases increase the false positives Dara generates. We make the observation that variables which determine program control flow are critical to a programs state, such variables are identified using static source code analysis. Post analysis false positives are fed back and used to identify key variables.

State Space Explosion Unique combinations of variable values compose unique states in an FSM. As such, each modelled variable expands the space of the FSM by the count of its unique logged values. We leverage the Z3 SMT solver as a program synthesis engine to collapse FSM state size as follows [3]. Pairs of nodes connected by edges are passed to Z3 as input-output examples of functions. If Z3 finds a set of operations which match all observed input-output examples, the unique instances of the variable are removed, and replaced with operation on edges which capture their state transfer. This drastically reduces FSM state space.

False Positives Inference with Z3 has the potential to generate incorrect operations if insufficient input-output examples are present in the logs. To detect, and reduce such errors we leverage Daikon to infer likely data invariants [4]. Each FSM node is composed of a set of log instances, we run Daikon on these instance to collect invariants on the node. Post model checking variable values are compared with collected invariants. If many invariants are invalidated (a high deviation in values from the collected logs), it is strong evidence that the inferred operations (or model itself) were incorrect. In such cases the model is recomputed including variables omitted from the prior instance of the model.

State Exploration Dara converts inferred models to Promella [6], and uses the SPIN model checker. Users supply a configuration of state invariants to SPIN. SPIN outputs traces of the approximate model that violate specified invariants.

Validation Model checking an approximate specification implies that flagged bugs may be false positives. Correlating inferred bugs with real bugs manually is challenging and laborious. A final contribution of our work is a functional mapping between inferred bugs, and real traces. Inferred bugs produce traces which can be scheduled and replayed using Dara's transparent model checker.

3 Prototype & Future Work

Our current prototype consists of 5K lines of Go, and includes facilities for generating FSMs, and generating likely traces with SPIN. We have applied Dara to a 200 line implementation of Dining philosophers and successfully identified fairness violations not present in existing logs. Our project lacks a transparent model checker, requiring that these bugs were verified manually. We are currently in the development of an open source checker for Go programs.

In the future we plan to extend our analysis to larger distributed systems with more complex properties. Our current techniques have only been applied to systems consisting of less than 500LOC symmetric systems.

Our target is to analyze, and find crucial bugs consensus algorithms mainly blockchain (BTCD [1]) and raft (ETCD [2]). Blockchain martians a large amount of distributed state in the from of transaction history which requires a precise model to simulate. Rafts algorithm maintains less state, but requires the precise modelling of a complicated protocol with nontrivial edge cases.

References

[1] BTCSUITE. an alternative full node bitcoin implementation. https://github.com/btcsuite/btcd, 2013.

[2] COREOS. Distributed reliable key-value store. https://github.com/coreos/etcd, 2013.

[3] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *TACAS 2008*.

[4] ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., AND XIAO, C. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* (2007).

[5] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP 2015*.

[6] HOLZMANN, G. J. The model checker spin. *IEEE Trans. Softw. Eng. 23*, 5 (May 1997), 279–295.

[7] LAMPORT, L., MATTHEWS, J., TUTTLE, M., AND YU, Y. Specifying and verifying systems with TLA+. In *EW 2002*.

[8] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI 2015*.

[9] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. Modist: Transparent model checking of unmodified distributed systems. In *NSDI 2009*, USENIX Association.

[10] ZALEWSKI. American fuzzy lop. http://lcamtuf.coredump.cx/afl, 2014.