Bandsaw: Log-Powered Test Scenario Generation for Distributed Systems

Ivan Beschastnikh Yuriy Brun Michael D. Ernst Arvind Krishnamurthy Thomas E. Anderson

Computer Science & Engineering
University of Washington

1 Introduction

Software testing is a widely used technique to eliminate defects and improve software quality. Testing is especially useful in the context of large distributed systems, which are notoriously difficult to reason about formally and remain out of reach for many existing analysis tools. A key problem in formulating a test case is identifying scenarios that can be (1) induced by a valid execution of the system under test, and are (2) different from the scenarios exercised by the suite of existing test cases. Today, this is done manually — a developer writes the code for a test case after considering the system implementation and the existing test suite. For a distributed system, this mental effort can be overwhelming since test cases are usually concurrent, involve multiple nodes and numerous message interleavings. Moreover, when coming up with a new test case, the developer usually focuses on the code artifacts and rarely, if ever, thinks about the abstract scenarios that the testing code represents.

The goal of our tool, *Bandsaw*, is to automate test *scenario* generation for distributed systems by considering the log of system's test suite executions. Bandsaw-generated scenarios are intended to be converted into test cases by the developer, which can then be run to test the implementation. Bandsaw, therefore, saves the developer the mental effort involved in coming up with a new test case. Test scenarios are also more abstract and simpler than the underlying testing code. These scenarios help developers to more easily reason about their test suites.

Bandsaw takes as input a console log file generated by the existing test suite, and outputs a new scenario. By construction, this scenario is (1) different from the ones already encoded in the test suite and (2) plausible — likely to be induced by a valid run of the system. Bandsaw generates scenarios by exploring different interleavings of concurrent events and stitching together previously observed scenarios at those points when the system is likely to be in a common state. The generated scenarios satisfy a set of automatically-mined, key temporal properties that are true of all the existing scenarios. The careful stitching of scenarios and the preservation of the mined properties make the resulting scenarios plausible.

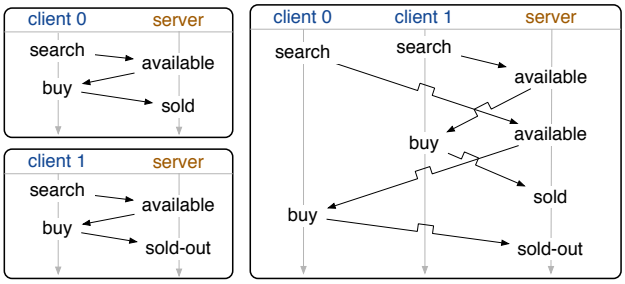

Figure 1: **(a)** A visualization of two input scenarios, extracted from a log of the system during the execution of its test suite. Each scenario is a Lamport space-time diagram in which time flows down, and events at each host are shown in a single column. **(b)** A test scenario output generated by Bandsaw based on the scenario inputs in (a).

Next, we illustrate how Bandsaw can be used to generate a useful test scenario for an example system that consists of a ticket selling server and clients who purchase tickets.

2 Motivation

Figure 1a shows two Lamport space-time diagrams of scenarios. These come from a testing log of a simple web-based application, in which a server maintains a limited number of tickets that clients search for and purchase. Each of the two tests that generated these scenarios involves a single client. The top execution in Figure 1a terminates in a successful sale, while the bottom one terminates with `sold-out`.

Figure 1b shows the Bandsaw-generated test scenario. This scenario features two clients and combines the two scenarios from Figure 1a into a new, concurrent scenario. This scenario is *plausible* and is not guaranteed to be possible. While Bandsaw-generated scenarios may not correspond to a valid system execution, the utility of Bandsaw is in relieving the developer from generating such scenarios by hand. The insight is that it is often easier for a developer to tell if a given scenario is valid than to come up with a new feasible scenario from scratch.

3 Bandsaw design

Bandsaw first parses the input console log generated by the system's test suite into input test scenario (e.g., Figure 1a). A test scenario is a partially ordered set of messages logged by the system. Next, Bandsaw mines a set of invariants that capture certain kinds of ordering between pairs of events generated by the system. These invariants are true for all the input test scenarios, and Bandsaw uses these as constraints on the kinds of test scenarios it can generate. Finally, Bandsaw uses counter-example guided refinement [3] to construct a model that (1) describes all the input scenarios, and (2) generalizes to scenarios that have not been observed. These predicted scenarios (e.g., Figure 1b) are then ranked according to the likelihood that they can be generated by the system, and then displayed to the user.

3.1 Temporal invariant mining

After extracting the scenarios from the input log, Bandsaw mines fives kinds of temporal invariants that describe certain types of orderings between pairs of events generated by the system [1]. For example, one invariant in the input log in Figure 1a is $\langle$*search@client* ***AlwaysFollowedBy*** *available@server*$\rangle$. This invariant relates the *search* event at the client with the *available* event at the server and indicates that whenever the *search* event occurs, the *available* event must also occur later in the same scenario. In addition to **AlwaysFollowedBy**, Bandsaw also mines the **NeverFollowedBy**, **AlwaysPrecedes**, **AlwaysConcurrentWith**, and **NeverConcurrentWith** invariants.

3.2 Modeling the input scenarios

Bandsaw uses a hyper-graph to model the input scenarios. This model is a generalization of the model used by Synoptic [2] — a tool for modeling totally ordered event sequences. Like Synoptic, Bandsaw uses a refinement procedure, which eliminates counter-example scenarios — scenarios from the hyper-graph model that violate at least one of the mined invariants. Refinement improves the model's accuracy, but it also increases the size of the model. Because of this, and because of the inherent complexity in traversing a hyper-graph, the resulting Bandsaw models are difficult to interpret. Therefore, unlike Synoptic, the goal of Bandsaw is not to display the final model to the user, but rather to show the user scenarios predicted by the model, which can then be used to generate test cases.

4 Discussion and related work

Bandsaw presents the user with a test scenario and the developer must manually write the corresponding test case invoking the scenario. This process could be automated by integrating Bandsaw-generated scenarios with the system's source code, and by using ideas similar to those in SherLog [11]. This is our future work.

Totally ordered sequences of events have been mined to automatically generate test cases [5, 4] and create models that help with, for example, object usage anomalies [9]. However, there is little work on mining executions of distributed systems. Kumar et al. [6] mine message sequence graphs, which resemble Bandsaw's scenarios. However, they do not use them for predicting new scenarios, which is the focus of Bandsaw.

More generally, logs from distributed systems have been used to detect system problems [8, 10], structural properties such as dependencies [7], and other features of the system. However, there is little prior work on leveraging log analysis to support developers of distributed systems during testing.

References

[1] BESCHASTNIKH, I., BRUN, Y., ERNST, M. D., KRISHNAMURTHY, A., AND ANDERSON, T. Mining Temporal Invariants from Partially Ordered Logs. In *Proc. of SLAML* (2011).

[2] BESCHASTNIKH, I., BRUN, Y., SCHNEIDER, S., SLOAN, M., AND ERNST, M. D. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proc. of FSE* (2011).

[3] CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided Abstraction Refinement. In *Computer Aided Verification* (2000), Springer, pp. 154–169.

[4] DALLMEIER, V., KNOPP, N., MALLON, C., HACK, S., AND ZELLER, A. Generating test cases for specification mining. In *In Proc. of ISSTA* (2010).

[5] FRASER, G., AND ZELLER, A. Exploiting common object usage in test case generation. In *In Proc. of ICST* (2011).

[6] KUMAR, S., KHOO, S.-C., ROYCHOUDHURY, A., AND LO, D. Mining Message Sequence Graphs. In *Proc. of ICSE* (2011).

[7] LOU, J.-G., FU, Q., WANG, Y., AND LI, J. Mining Dependency in Distributed Systems through Unstructured Logs Analysis. *SIGOPS Oper. Syst. Rev. 44* (March 2010), 91–96.

[8] LOU, J. G., FU, Q., YANG, S., XU, Y., AND LI, J. Mining Invariants from Console Logs for System Problem Detection. In *Proc. of ATC* (2010).

[9] WASYLKOWSKI, A., ZELLER, A., AND LINDIG, C. Detecting Object Usage Anomalies. In *Proc. of FSE* (2007).

[10] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting Large-Scale System Problems by Mining Console Logs. In *Proc. of SOSP* (2009).

[11] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *Proc. of ASPLOS* (2010).