Learning-based Testing of Distributed Microservice Architectures: Correctness and Fault Injection

Karl Meinke and Peter Nycander

KTH Royal Institute of Technology, Stockholm 100-44, Sweden,
karlm@kth.se, peternyc@kth.se,

Abstract. We report on early results in a long term project to apply *learning-based testing* (LBT) to evaluate the *functional correctness of distributed systems* and their *robustness to injected faults*. We present a case study of a commercial product for counter-party credit risk implemented as a distributed microservice architecture. We describe the experimental set-up, as well as test results. From this experiment, we draw some conclusions about prospects for future LBT tool research.

Keywords: automated test case generation, fault injection, learning-based testing, microservice, requirements testing, robustness testing,

1 Introduction

1.1 Overview

Functional testing of distributed systems presents one of the greatest challenges to any test automation tool. Significant execution problems exist, such as: system latency for individual test cases, global state reset, non-determinism leading to unrepeatable errors, and the existence of faults in communication infrastructure, to name but a few.

Learning-based testing [5] (LBT) is an emerging paradigm for fully automated black-box requirements testing. The basic idea of LBT is to combine machine learning with model checking, integrated in a feedback loop with the system under test (SUT). An LBT architecture iteratively refines and extends an initial model of the SUT by incremental learning, using test cases as learner queries. During this process, it model checks for violation of user requirements. Current LBT technology can construct and judge thousands of test cases per hour (depending on SUT latency), which is potentially useful both for testing complex distributed systems and for fault injection.

1.2 Problem Formulation

We consider the problem of applying LBT to *black-box testing the functional correctness of distributed systems* and *testing their robustness to injected faults*. We

describe an experiment to perform requirements testing and fault injection on a *distributed microservice architecture* for counter-party credit risk analysis known as triCalculate. This application is a commercial product developed by TriOptima AB for the Over-The-Counter (OTC) derivatives market. Our experiment had several goals, including an evaluation of:

1. ease of formal modeling of correctness and robustness requirements,
2. ease and efficiency of fault injection and test case tear-down in a distributed system through the use of LBT wrapper constructs
3. success in detecting SUT errors using low fidelity inferred models.

Furthermore, our experiments were made using an existing tool LBTest [6], which lacks any optimisation for distributed system testing. Thus an additional goal was: 4. to evaluate what architectural changes could be made to LBTest, that might improve its performance in this context. In Section 5 we make some suggestions for future tool development.

2 Background

2.1 Microservice Architectural Style

The microservice architectural style [4] implements a single application as a suite of many small services that communicate using a language agnostic mechanism such as HTTP. The style is a new approach to Service Oriented Architecture (SOA). The benefits of this style include: technology independence, resilience, scalability and ease of deployment. In triCalculate, network communication is event-based, using RabbitMQ.

2.2 Fault Injection

Fault injection (FI) has traditionally been deployed to evaluate the robustness of software in the presence of hardware faults (such as bit-flip errors). A classification in [8] includes: hardware implemented (HIFI), software (SIFI) and model implemented (MIFI). Our approach here is closest to SIFI, but LBT can also support MIFI, through its model inference. A characteristic of fault injection is the combinatorial growth of faults [1], and large test suites are typically needed. The potential of LBT to rapidly generate test cases may therefore be beneficial in this context. Interesting high-level faults that were considered to inject into triCalculate included: restarting services, starting up several service instances, communication faults, and killing service instances.

[1] A *fault* is a triple consisting of a type, a location and a time [8].

2.3 Learning-based Testing

For this experiment, we used LBTest version 1.3.2 [6] which is an LBT tool for testing reactive systems. The architecture of LBTest is illustrated in Figure 1. It has previously been successfully used to test monolithic financial applications [3]. Reactive systems are modeled and learned as deterministic finite state machines, specified using propositional linear temporal logic (PLTL), and model checked with NuSMV [2]. LBTest has a modular architecture to support a variety of learning algorithms, model checkers and equivalence checkers.

Noteworthy in Figure 1 is the *communication wrapper* around the SUT, which is responsible for communication and data translation. Wrappers are also responsible for test set-up and tear-down between individual test case executions. These activities can be problematic for distributed systems, requiring complex network management and reset actions. Furthermore, wrappers support the abstraction of infinite state systems into finite state models, through data partitioning. Using the same data abstraction principles, wrappers can also be used to support fault injection. However, to date, there has been no research published on LBT for robustness testing. A more general open question is how to extend LBT to different types of distributed systems, which is a long-term goal of the project [9].

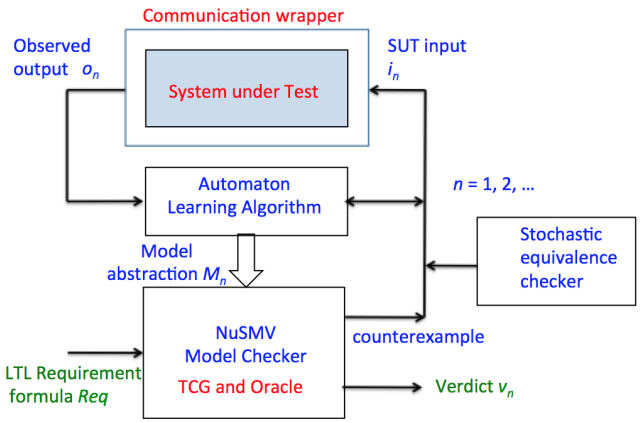

Fig. 1. LBTest architecture

3 Experimental Set-up

The SUT triCalculate calculates counter-party credit risks using distributed microservices to host different parts of the calculation. It consists of about 100KLoC. Figure 2 illustrates the triCalculate architecture in terms of separate microservices and their intercommunication.

The system has just one use case: *a user uploads files and eventually receives output.*

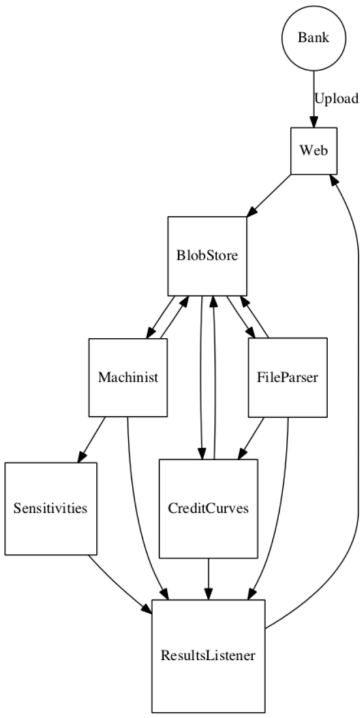

Fig. 2. triCalculate microservice architecture

To manage service scheduling and concurrency, we took a grey-box testing approach, by opening up the RabbitMQ messaging system to the LBTest wrapper. In particular, the wrapper made non-trivial use of the RabbitMQ API in order to efficiently and reliably tear down inter-service message queues after each test case execution. In [3] we have also shown that efficient tear-down is an important step to fully exploit the high test throughput of LBT methods.

The most relevant correctness property to be tested is that *the end-to-end calculation finishes in a successful way*, i.e. when a file is uploaded then sometime in the future results will be published. Such end-to-end requirements for microservices are quite difficult to test by traditional (i.e. non-formal, manual) methods. We also focused on a single fault scenario: *SUT restart*, to verify that triCalculate handles system crashes and recovery in a correct way. To test the correct execution of each use case step, the relevant symbolic input values from LBTest to the SUT were `continue` and `restart`.

The SUT state variables for each observation were obtained from database entries in the central storage unit. These state variables were returned by the wrapper as 6 symbolic output variables, each of which flagged the completion of one use case step (i.e. calculation step). Using these output variables, we could express each of the 6 *sunny day use case steps*, e.g. step 1:

```
G(session = session_none & input = continue ->
X(session = session_ok))
```

This sunny day use case model then had to be extended with alternative *rainy day* use case steps, to model SUT robustness under `restart`, e.g. step 1.b

```
G(session = session_none & input = restart ->
X(session = session_none))
```

To continuously monitor the internal SUT status, an additional output `warning` variable was used, specified by: `G(warning = warning_none)`

4 Test Results

In benchmarking LBT methods, our main arbiter of success is the capability to identify known errors (i.e. injected mutations) as well as previously unknown SUT errors within a reasonable time frame. Since LBT is a fully automated testing approach, it is feasible to run a tool such as LBTest for several hours.

A secondary measure of testing success is the number of test cases (queries) executed and the size of the inferred state machine model. However, LBT methods always produce a finite abstraction of the SUT, which is itself an infinite state system. It can be difficult to predict the size of this abstraction. Therefore besides its absolute size, the degree of convergence of the learned model is another important test performance indicator. Convergence is measured indirectly through stochastic equivalence checking.

Note that test suite size and inferred model size are both heavily influenced by the choice of learning algorithm. Since LBT is still an emerging technique, it is instructive to experiment with different learning algorithms and compare their performance on the same SUT.

To begin our testing experiment, ad-hoc SUT mutations were injected and successfully detected. These helped to confirm the correct construction of both the user requirements and the wrapper.

Testing was then performed on the unmodified SUT using two different automata learning algorithms: IKL [5] and L*Mealy [7]. L*Mealy is a straightforward generalisation of Angluin's well-known L* algorithm [1] to deal with multi–valued output alphabets. The two algorithms have similar asymptotic complexity properties. IKL is an incremental learning algorithm that produces hypothesis automata with much greater frequency than L*Mealy. It is well adapted to testing systems that are too big to be completely learned. The heuristic used for stochastic equivalence checking was *first difference*, which is the only heuristic that is practically feasible for SUTs with long test case latency.

Using IKL, a testing session was conducted for 4 hours and 3 minutes. During this time LBTest made a total of 139 queries leading to 7 warnings. A 24 state

model was learned after 16 iterations. Using L*Mealy, a testing session was conducted for 7 hours and 32 minutes. During that time LBTest made a total of 280 queries leading to 2 warnings. A 38 state model was produced after 3 iterations.

In this case study, IKL is slightly more efficient requiring 5.8 queries per learned state versus L*Mealy's 7.4 queries. Furthermore, IKL produces about 5 times as many hypothesis automata as L*Mealy. Thus, IKL allows the model checker to detect a greater number of SUT failures.

The final model produced by L*Mealy was slightly better converged than the final model produced by IKL, as measured by stochastic equivalence checking with the SUT. However, this improvement was marginal given that twice as many queries were used. In fact this small difference suggests that the models produced by both learning algorithms were still highly non-converged.

In this case study, the use-case structure of our PLTL user requirements allowed us to measure *requirements coverage* in a precise way using graph coverage. This approach is similar to the requirement coverage model of [10]. Here the results were quite positive. Every simple path through the use case is covered by at least one path through the final learned model, in both the IKL and L*Mealy inferred models.

The latency time of the triCalulate SUT was relatively long, on average 1.6 minutes per test case. Therefore, the inferred models have quite low fidelity (convergence), based on a small set of samples. This latency could be attributed primarily to the computationally intensive SUT, and not to the overhead of machine learning, model checking or network communication.

5 Conclusions and Future Research

Regarding experimental goal 1, we confirmed the viability of modeling correctness and robustness requirements in a single use-case model, consisting of sunny and rainy day scenarios. These requirements could be formalised using propositional linear temporal logic by a test engineer (Nycander) with no previous experience of using temporal logic.

Regarding goal 2, fault injection was successfully modeled at a high-level of abstraction using symbolic input data types, while the actual semantics of fault injection could be efficiently implemented in a wrapper construction. This approach to robustness testing seems powerful and flexible, but it requires the test engineer to make correct and sometimes innovative wrapper constructions.

Regarding goal 3, the performance of LBTest compares favourably with current manual techniques to perform end-to-end testing. LBTest was able to discover multiple failed test cases within a reasonable time frame. Although the inferred models had low convergence, a high level of user requirements coverage was nevertheless achieved.

With respect to goal 4, the non-determinism exhibited by triCalculate showed that new approaches are necessary to improve the inference and modeling of distributed systems. Adapting automata learning algorithms to non-deterministic

SUTs is technically straightforward. However, problems arise with measuring convergence of learning, since some non-deterministic behaviours may have very low probability of occurrence (unrepeatable errors).

The problem of poorly converged models is primarily due to high test case latency, which in this case arises from a computationally intensive SUT more than from network communication. The general situation for other microservice architectures merits further investigation. However, we can deduce that improving the coverage of distributed systems (as measured by learned state space size) is one of the most important and challenging problems for LBT tools.

One possible approach might be to distribute learning across the network. We could try to infer a set of local automaton models that can be combined by asynchronous parallel composition into a single global model. This approach might be well–suited to microservice architectures, since using microservers should reduce the difficulty of learning the individual local models. Another approach might be to try to re-use inferred models from unit and integration testing within system testing. These two approaches may even be mutually compatible.

K.Meinke wishes to thank VINNOVA and Scania AB for financial support of this research within [9], and R. Svenningson for valuable discussions on fault injection. P. Nycander wishes to thank TriOptima AB for hosting his Masters thesis research.

References

1. D. Angluin, Learning regular sets from queries and counterexamples, Information and Computation, 75:87-106, 1987.
2. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella: NuSMV 2: An OpenSource Tool for Symbolic Model Checking, in Proc. Int. Conf. on Computer-Aided Verification (CAV 2002), 2002.
3. L. Feng, S. Lundmark, K. Meinke, F. Niu, M.A. Sindhu, P.Y.H. Wong: Case Studies in Learning-based Testing, pp. 164-179 in Proc. 25th IFIP Int. Conf. on Testing Software and Systems (ICTSS 2013), Springer LNCS 8254, 2013.
4. M. Fowler, Microservices, `http://martinfowler.com/articles/microservices.html`, 2014.
5. K. Meinke and M. Sindhu: Incremental Learning-Based testing for Reactive Systems, pp 134-151 in: Proc. Int. Conf. on Tests and Proofs TAP 2011, LNCS 6706, 2011.
6. K. Meinke and M. Sindhu: LBTest: A Learning-based Testing Tool for Reactive Systems, in Proc. Sixth Int. IEEE Conf. on Software Testing, Verification and Validation (ICST-2013), 2013.
7. O. Niese. An Integrated Approach to Testing Complex Systems. PhD thesis, University of Dortmund, 2003.
8. R Svenningson: Model Implemented Fault-injection for Robustness Assessment. Licentiate Thesis, KTH Stockholm, 2011.
9. VINNOVA FFI project, VIRTUES (Virtualized Embedded Systems for Testing and Development), `http://www.csc.kth.se/~karlm/virtues/`
10. M. Whalen, A Rajan, M. Heimdahl, S. Miller: Coverage Metrics for Requirements-Based Testing, pp 25-35 in: Proc Int. Symp. on Software Testing and Analysis, ISSTA 2006, ACM Press.