Towards A Scalable Cloud Platform for Search-Based Probabilistic Testing

Louis M. Rose*, Simon Poulding*, Robert Feldt† and Richard F. Paige*
*Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK
{louis.rose,simon.poulding,richard.paige}@york.ac.uk
†School of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden
robert.feldt@bth.se

***Abstract*—Probabilistic testing techniques that sample input data at random from a probability distribution can be more effective at detecting faults than deterministic techniques. However, if overly large (and therefore expensive) test sets are to be avoided, the probability distribution from which the input data is sampled must be optimised to the particular software-under-test. Such an optimisation process is often resource-intensive. In this paper, we present a prototypical cloud platform—and architecture—that permits the optimisation of such probability distributions in a scalable, distributed and robust manner, and thereby enables cost-effective probabilistic testing.**

I. Introduction

The probabilistic generation of test inputs by random sampling from an input profile (i.e. a probability distribution over the input domain of the software-under-test) has two significant advantages over deterministic test inputs. Firstly, if the software is repeatedly tested using the same deterministic test set, it can become 'overfitted' to the tests: while the software may operate correctly for the specific inputs specified by the deterministic test set, it may still exhibit faults when run using any other input. Using randomly-generated inputs reduces such overfitting. Secondly, it is relatively straightforward to construct a test set of any given size using a probabilistic approach—additional test inputs may be generated simply by taking further samples from the input profile—and so adapt the amount of testing to the time and budget available.

However, probabilistic test generation using an *arbitrary* input profile, such as a uniform distribution over the input domain, is unlikely to be cost-effective. For example, if a component of the software is only exercised when the software is executed with inputs from a small region of the input domain, test inputs sampled from a uniform distribution will rarely exercise that component. Therefore a large set may be required to detect faults in that component, but large test sets are not cost-effective: each test case requires the application of an 'oracle' to check the correctness of the observed output from the software and many oracles, such as a test engineer interpreting a specification document, are expensive. Instead, the input profile must be optimised to the specific software-under-test if probabilistic testing is to be cost-effective.

Poulding [1] has previously demonstrated an automated algorithm for optimising profiles in this way that uses a criterion developed by Thévenod-Fosse and Waeselynck [2] based on structural coverage: maximise the probability that any given part (e.g. statement or branch) of the software-under-test is exercised by one test input sampled at random from the profile. Poulding's algorithm uses a metaheuristic optimisation (or 'search') method: iterative hill-climbing. At each iteration, small changes are made to the current input profile to create a number of candidate profiles, the candidate profiles are evaluated against Thévenod-Fosse and Waeselynck's criterion to determine their 'fitness', and the fittest of these candidates becomes the current input profile in the next iteration.

Since each candidate profile is a probability distribution, it is evaluated by executing the software-under-test with multiple inputs sampled from the distribution. These executions are not explicitly part of the testing processes, and the (potentially expensive) oracle is *not* applied to the results; only after the input profile has been optimised by the algorithm is a small test set generated from the profile and the oracle applied to the results from this test set. Nevertheless, Poulding's search-based algorithm can be resource-intensive and to ensure practicality of this testing strategy, it is beneficial to derive an optimal input profile as quickly as possible.

It is this challenge that we address in this paper. We present a prototypical implementation of a cloud platform that optimises input profiles for probabilistic testing quickly and robustly, with the objective of scaling the testing approach to larger software. The paper makes the following contributions:

- A probabilistic testing platform that derives optimised input profiles, available at http://coco.herokuapp.com.
- The design of a service-oriented architecture for integrating a testing platform with the software-under-test in a robust and scalable manner.
- An evaluation of the platform with a realistic case study: testing a model transformation used in specifying the behaviour of a robot.

II. Design and Implementation

We have designed and implemented a probabilistic testing platform, *coco*, that can quickly optimise input profiles. Our platform employs a service-oriented architecture to achieve scalability and robustness. Figure 1 outlines our approach. A user of the platform provides a URL for the software that they wish to test. The testing platform requests, via the user-provided URL, an initial input profile from a testing harness that wraps the software-under-test (SUT), and then proceeds to derive an optimised input profile by application of Poulding's automated search-based algorithm. The process of evaluating the fitness of a candidate profile involves executing the SUT

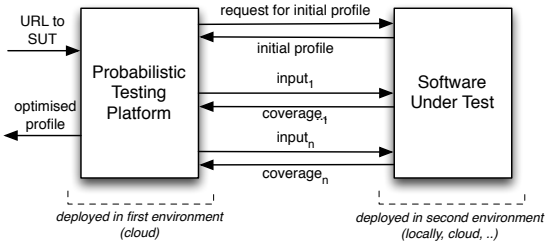

Fig. 1. An overview of our proposed architecture. Boxes indicate software components and arrows indicate communication via HTTP.

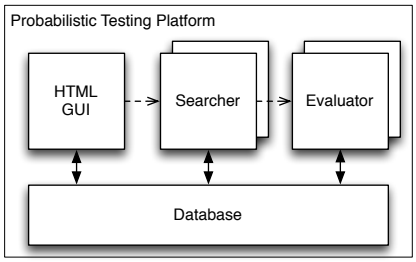

Fig. 2. The architecture of our testing platform. Boxes indicate software components, which communicate via message queues (dashed arrows) and with a database via HTTP (solid arrows). In the configuration depicted above, we have scaled out search and evaluation by deploying additional instances of those components.

using multiple inputs sampled from the profile. The SUT is instrumented to determine which parts of the software are exercised by each input, and this coverage data is returned to the testing platform in order to calculate the candidate profile's fitness.

Key to our architecture (Figure 1) is the decoupling of the testing platform from the SUT via HTTP, which provides several benefits. The testing platform and SUT can be implemented in different programming languages—allowing us to support the testing of programs written in arbitrary languages—and in separate environments (e.g. different machines)—allowing us to tune and scale each environment independently. In addition, the testing platform is more robust with respect to failures that arise in the SUT: candidate evaluation can be re-tried or aborted if the SUT fails to respond.

Further decoupling at the level of data representation is achieved through the use of a stochastic grammar to represent a distribution over the (potentially highly-complex) input domain of the SUT; this technique is described in [3]. The SUT wrapper specifies this grammar as the initial profile. The testing platform provides strings of tokens sampled from candidate profile grammars that are interpreted by the wrapper as inputs to the SUT. Consequently, the testing platform and SUT do not need to share the same data representation.

Internally, our testing platform comprises three components that communicate via message queues (Figure 2). The *HTML GUI* allows users of the platform to start new probabilistic testing tasks, to view the status of existing tasks, and to download the optimised input profile from which they can sample random test data. The *Searcher* encapsulates Poulding's automated search-based algorithm and, during the optimisation process, identifies candidate input profiles to evaluate. The *Evaluator* encapsulates the logic required to communicate with the SUT to make this evaluation.

The *HTML GUI*, *Searcher* and *Evaluator* components are decoupled via message queues and a shared database. For example, when the *Searcher* requires an evaluation to be performed it creates a record for that evaluation in the shared database, places a request for evaluation work on the message queueing system and polls the database to determine when the evaluation has completed. Meanwhile, the *Evaluator* dequeues requests for evaluation work, interacts with the SUT, and updates the evaluation record in the database with the coverage data obtained from the SUT.

Decoupling the components of the testing platform via message queues and a shared database provides benefits that are analogous to those described for the architecture as a whole: the components can be implemented in different languages, can be deployed in separate environments, and can be scaled out independently by instantiating additional instances of a component. Additionally, the use of message queuing increases the robustness of the testing platform as whole, because (search and evaluation) jobs that fail can be re-queued and re-tried.

Our testing platform has been implemented in Ruby and uses a number of off-the-shelf libraries. The *HTML GUI* uses the Ruby on Rails web framework and the *Evaluator* uses the HTTParty HTTP client for Ruby. The *Searcher* component is written in JRuby[1] and Java. Message queuing is implemented with the Resque library. We selected Ruby and Ruby libraries due to our familiarity with deploying and scaling out Ruby applications on the Heroku cloud platform. For the *Searcher* component, we selected JRuby to allow us to more readily experiment with alternative Java libraries for metaheuristic search algorithms (such as ECJ[2] and JMetal[3]) in future work.

III. Example and Experience Report

We report here our experiences of applying our probabilistic testing platform to derive an optimised input profile for a program written in a model transformation language (introduced below), and reflect on the benefits and drawbacks of our platform compared to the ad hoc and naïve architecture used in our previous work on the same program [4]. We present empirical evidence to demonstrate performance improvements over the testing performed in our previous work, and also reflect on our experiences with using the probabilistic testing platform for the preliminary experimentation presented below.

Model transformations are programs that consume a source model and produce a target model. The models on which model transformations operate must conform to a modelling language (metamodel) that specifies the structures from which valid instance models can be constructed. For example, UML is a metamodel, a class diagram is a model, and a program that consumes a class diagram and produces a database schema is a

[1] An implementation of Ruby that runs on the Java Virtual Machine
[2] http://cs.gmu.edu/~eclab/projects/ecj/
[3] http://jmetal.sourceforge.net

model transformation. Model transformation is key to model-driven engineering and is supported by model transformation languages such as XSLT [5], Query/View/Transformation [6] and the Atlas Transformation Language [7]. As Mottu et al. discuss in [8], when the output of the SUT is a model—as it is for a model transformation—applying an oracle can be particularly difficult: not only is the output itself complex in nature, but the observed and predicted models may be syntactically different even when they are semantically equivalent. There is, therefore, a particular motivation to use probabilistic testing to derive an *optimised* input profile that minimises the size of the test set required to exercise the model transformation and hence the costs involved in applying the oracle.

A. The Software-under-Test

Our previous work applies probabilistic testing to a model transformation for a Lego Mindstorms robot [4]. For this transformation, the source and target models conform to domain-specific modelling languages that describe a high-level plan for the robot and a low-level strategy for executing the plan, respectively. The transformation automatically derives a strategy from a plan. Previously, we were able to derive an optimised input profile but experienced several issues with the ad hoc and naïve architecture we used at the time:

- Deriving the optimised input profile took almost 17 hours, averaging approximately 0.3 seconds per execution of the model transformation (which equates to 240,000 executions in total).
- For each concurrent instance of the technique that we ran for the empirical work, we had to provision dedicated (virtual) machines for the search-based algorithm and for the SUT. Provisioning the machines was a significant infrastructural overhead, and limited the extent to which we could conduct our empirical work.
- The search-based algorithm did not store its state in a persistent store. Consequently, failures on the machines running the algorithm were catastrophic, and destroyed any progress made towards deriving an optimised input profile.
- The search-based algorithm and SUT were tightly coupled, which was problematic when, for instance, the performance of the SUT degraded because it was impossible to re-provision the SUT without aborting the search. Performance degradation of the SUT is a particular risk in applying our approach because of the random nature of the input data, which might be very different to the kinds of input against which the software has previously been tuned.

The Lego Mindstorms model transformation was implemented in the Epsilon Transformation Language [9], a model transformation language written in Java. In our previous work, we constructed a light-weight adapter for the Lego Mindstorms transformation that facilitated probabilistic testing. The wrapper allowed us to deploy the Lego Mindstorms transformation as a web application on the Heroku cloud platform. Scalability was achieved by creating multiple copies of the application, and having each instance of the search-based algorithm address a unique copy of the transformation application.

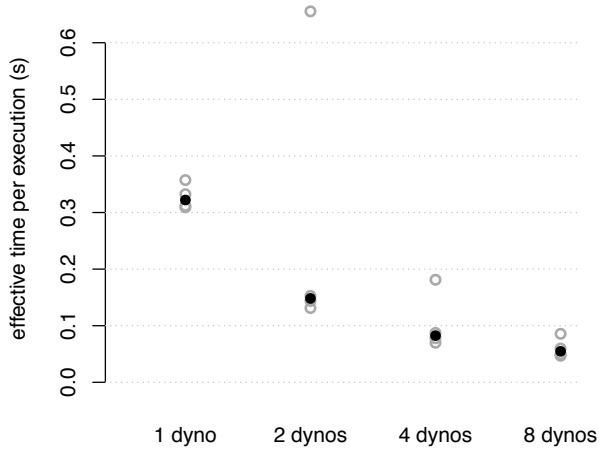

Fig. 3. Scatter graph showing the change in performance with the number of dynos. The open circles are the individual performance measurements, and the filled circles are the median performance of each configuration.

For this paper, we adapted the architecture presented in Figure 2 for the testing platform to provide a similarly scalable application for executing the Lego Mindstorms (or other similar) model transformation. This application fulfils the role of the wrapper of the software-under-test in the architecture of Figure 1.

B. Empirical Evaluation

The empirical evaluation focuses on the major objective of our proposed testing platform: that of improved performance through horizontal scaling.

Both the testing platform and transformation application (the software-under-test) were deployed using the Heroku cloud platform. The testing platform was run in four different configurations which made available 1, 2, 4, and 8 dynos (Heroku virtual machines). Each dyno supports one *Evaluator* instance, and so the 2 dyno configuration, for example, could run two *Evaluator*s concurrently during the search. In each configuration, the equivalent number of processing dynos were available to the transformation application to support the increased load from the *Evaluator*s. The time taken for 5 iterations of the search algorithm was measured (a total of 7550 executions of the model transformation by inputs sampled from candidate profiles), and this time divided by 7550 to calculate an effective time per execution. In order to reduce the impact of wide variation in performance as a result of the time-shared Heroku platform (see below), the performance of each configuration was measured 4 times.

The results are shown in Figure 3. We note that the median performance (filled circle) for the 1 dyno configuration is similar to the 0.3 seconds achieved by the ad hoc architecture used in our previous work; this is to be expected given that the 1 dyno configuration employs similar computing resources to that previous architecture. However, as the number of dynos is increased, the algorithm performance improves. The median performance achieved with 8 dynos is 0.055 seconds

per execution: over 5 times faster than using the architecture of our previous work. This result illustrates the performance improvements that are possible due to the horizontal scaling of our testing platform's architecture.

C. Reflective Evaluation

In addition to the performance improvements described in the previous section, we now reflect on additional, unexpected benefits that we have experienced in applying our probabilistic testing platform to test the Lego Mindstorms transformation described above.

In performing our experimentation we observed non-trivial variation in execution time both for the testing platform and the software under test: see, for example, the very slow outlier for the 2 dyno configuration in Figure 3. We attribute this to deployment on the Heroku cloud platform which implements a time-sharing policy: our applications execute on a machine shared with the applications of other users. When other applications deployed on the same machine experience high load, our applications are likely to receive less CPU time. Using message queueing allows us to potentially distribute load over a number of threads deployed on different machines, reducing the overall chance of our applications being affected by contention for CPU time on a single machine.

Furthermore, the architecture that we propose in this paper is amenable to dynamic scaling in which the number of threads allocated to a task (e.g. evaluating candidate input profiles) can be automatically increased in response to a backlog. By monitoring the queue size, we can approximate the load on the system and instantiate additional—or re-allocate existing—computational resources to reduce bottlenecks.

Finally, a downside of the architecture that we proposed in this paper is that the asynchronous interface between the testing platform and the software under test is implemented using a simplistic polling mechanism which introduces additional load on the software-under-test (which must provide information about the status of executions) and increases latency (as there is often a delay between an execution completing and the testing platform sending a request to obtain the results). We will investigate introducing a callback mechanism to address these issues, and further reduce the time taken to derive an optimal input profile.

IV. Related Work

We briefly describe examples of other testing tools that have employed cloud architectures in order to scale automated algorithms to real-world software. Oriol and Ullah [10] describe a cloud-deployed version of the automated random testing tool YETI[4] motivated by the need to accommodate software-under-test that executes relatively slowly, and to isolate potentially damaging executions of the software from the main testing mechanism. Di Geronimo et al. [11] propose a search-based technique for generating JUnit test cases using a genetic algorithm parallelised using Hadoop MapReduce; the motivation is to enable the technique to leverage high-performance parallel computing environments such as cloud computing and GPU cards. We note that in common with the search-based technique of Di Geronimo et al., our use of a cloud architecture is for the *generation* of test cases, in contrast to YETI where the cloud resources are applied to the large-scale *execution* of test cases.

[4]YETI differs from Poulding's algorithm in that it targets faults identified by 'automatic' oracles such as precondition violations and unhandled exceptions.

V. Conclusions

Poulding's probabilistic testing algorithm, like many other techniques for automated testing, is a promising—but computationally expensive—approach. We argue that scalable, distributed and robust architectures are crucial for cost-effective application of automated testing techniques, particularly those based on search algorithms. We have presented a preliminary version of our probabilistic testing platform, described its underlying service-oriented architecture, and evaluated the platform by applying it to test a model transformation. Our initial experience with the platform and its architecture indicate that is more scalable and robust than our previous ad hoc and naïve implementation. In future work, we will seek to also demonstrate the generality of our platform by applying it to test further examples of model transformations, and to test programs written in other programming languages.

Acknowledgement

This work was funded in part by EPSRC grant EP/J017515/1, DAASE: Dynamic Adaptive Automated Software Engineering.

References

[1] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *IEEE Trans. Software Eng.*, vol. 36, no. 6, pp. 763–777, 2010.

[2] P. Thévenod-Fosse and H. Waeselynck, "An investigation of statistical software testing," *J. Software Testing, Verification and Reliability*, vol. 1, no. 2, pp. 5–26, 1991.

[3] S. Poulding, R. Alexander, J. A. Clark, and M. J. Hadley, "The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing," in *Proc. Genetic and Evolutionary Computation Conf. (GECCO)*, 2013, (to appear).

[4] L. Rose and S. Poulding, "Efficient probabilistic testing of model transformations using search," in *Proc. CMSBSE workshop, co-located with Int'l Conf. Software Engineering*. IEEE / ACM, 2013, (in press).

[5] W3C, "XSL Transformations (XSLT) Specification V1.0 [online]," [Accessed 21 June 2013] Available at: http://www.w3.org/TR/xslt, 1999.

[6] OMG, "Query/View/Transformation Specification V1.1 [online]," [Accessed 21 June 2013] Available at: http://www.omg.org/spec/QVT/1.1/, 2011.

[7] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Proc. Satellite Events at the Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS)*, ser. Lecture Notes in Computer Science, J.-M. Bruel, Ed., vol. 3844. Springer, 2005, pp. 128–138.

[8] J.-M. Mottu, B. Baudry, and Y. Traon, "Model transformation testing: oracle issue," in *Proc. Workshops at Int'l Conf. Software Testing Verification and Validation*, April 2008, pp. 105 –112.

[9] D. Kolovos, R. Paige, and F. Polack, "The Epsilon Transformation Language," in *Proc. ICMT*, ser. LNCS, vol. 5063. Springer, 2008, pp. 46–60.

[10] M. Oriol and F. Ullah, "YETI on the cloud," in *Proc. Workshops of the Int'l Conf. on Software Testing, Verification, and Validation*, 2010, pp. 434–437.

[11] L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro, "A parallel genetic algorithm based on Hadoop MapReduce for the automatic generation of JUnit test suites," in *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, 2012, pp. 785–793.