POSTER: A Test Infrastructure for Self-Adaptive Software Systems

Eric Kilmer, Timothy Braje, Dinara Doyle, Tim Meunier, Philip Zucker, Jeffrey Hughes,
Michael Depot, Mark Mazumder, George Baah, Karishma Chadha, Robert Cunningham
Lincoln Laboratory, Massachusetts Institute of Technology
244 Wood Street
Lexington, Massachusetts 02420–9108
{eric.kilmer,tbraje,timothy.meunier,philip.zucker,jeffrey.hughes,mike.depot,mazumder,rkc}@ll.mit.edu

***Abstract*—Today's software often outlives the hardware for which it was initially designed. Autonomous systems in particular are deployed into scenarios where resources change and sensors degrade. In DARPA's Building Resource Adaptable Software Systems (BRASS) program, eight performers, with three challenge problems each, must develop software systems that dynamically reconfigure in response to a perturbation in the operating environment to successfully recover and continue normal operation. There are a variety of systems ranging from autonomous robots to embedded systems sensors to mobile and distributed systems. Leveraging past programs as experience [1, 2] and robust testing techniques [3], we develop a generic infrastructure for testing and evaluating all of these systems.**

Our design and implementation of the testing infrastructure leverages abstraction and automation. We generically generate tests for each system using performer-defined parameters and values with the ability to direct future tests to specific corners of the parameter-value space. Using a safe, Haskell-derived REST API [4] to communicate, tests are driven through a test harness and sent to the test adaptor which interfaces with the system-under-test. Since each system behaves differently, we factor out common methods of the API into reusable components to produce system-agnostic interaction models that can be easily adjusted to system-specific features on a case-by-case basis. These interaction models and communication endpoints are developed in a type-safe way such that each challenge problem and test case are typed; this prevents run-time errors by preventing challenge problem logic and parameter mismatches. The results are then rolled up into verdict graphs to visualize each scenario's outcome and link to the concrete arguments that were used.

With 24 different challenge problems, the need for organization and abstraction is paramount. To prepare for the final code delivery after almost a year of development time, we used three risk reductions to 1) agree on a set of API messages and their fields, 2) define reproducible execution environments using Docker for Amazon Web Services, and 3) ensure the systems execute a full test case using their defined interaction model. We found that performers were constantly changing their parameter fields and value ranges without necessarily updating the documentation, which made API synchronization tedious and sometimes error-prone. In fact, every performer made API changes up to the deadlines of all risk reductions plus the final code drop. In addition to reviewing the API changes four times throughout Phase 2, we manually performed integration tests to verify that the new changes were incorporated accurately.

For future tests and integration, we recommend using a standard, language-agnostic API specification, like the OpenAPI Specification[1], where data types and REST endpoint implementations can be generated and synchronized accurately. Furthermore, the creation or existence of continuous integration tests can pinpoint regressions in the end-to-end interactions. If we want to go even further, the development or use of a language-agnostic state machine specification can enabled verifiable descriptions and implementation for each system's interaction model.

We found that GitHub provided a sufficient asynchronous platform for opening/resolving issues and questions regarding the systems and test harness. This allowed everyone to point exactly where in the code a question regarding implementation could be referenced. However, for more complex issues, face-to-face or phone call interactions provided the best turnaround and least ambiguity for issue resolution. When deploying performer's systems, Amazon's Elastic Container Service (ECS) allowed us to efficiently manage and scale the number of tests run during evaluation. Using docker containers meant that we could quickly execute smoke-tests locally, and once our smoke-tests passed, we were confident that, given the correct amount of resources, we could execute a large number of tests at scale on ECS. With respect to our development experience, the strong, statically typed compiler for Haskell, The Glasgow Haskell Compiler, afforded us a pleasant development environment where we could safely integrate new features and refactor with ease and confidence.

Approved for public release: distribution unlimited. This material is based upon work supported by the Defense Advanced Research Projects Agency under Air Force Contract No. FA8702–15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

REFERENCES

[1] L. M. Rossey, R. K. Cunningham, D. J. Fried, J. C. Rabek, R. P. Lippmann, J. W. Haines, and M. A. Zissman, "Lariat: Lincoln adaptable real-time information assurance testbed," in *Proceedings, IEEE Aerospace Conference*, vol. 6, 2002, pp. 6–2671–2676, 6–2678–6–2682 vol.6.

[2] C. V. Wright, C. Connelly, T. Braje, J. C. Rabek, L. M. Rossey, and R. K. Cunningham, "Generating client workloads and high-fidelity network traffic for controllable, repeatable experiments in computer security," in *Recent Advances in Intrusion Detection*, S. Jha, R. Sommer, and C. Kreibich, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 218–237.

[3] S. Peisert and M. Bishop, "How to design computer security experiments," in *Fifth World Conference on Information Security Education*, L. Futcher and R. Dodge, Eds. Boston, MA: Springer US, 2007, pp. 141–148.

[4] M. Mazumder and T. Braje, "Safe client/server web development with haskell," in *2016 IEEE Cybersecurity Development (SecDev)*, Nov 2016, pp. 150–150.

[1] https://www.openapis.org/