Extensible Testing for Infrastructure as Code

David Spielmann
david.spielmann@unisg.ch
University of St. Gallen
Switzerland

Daniel Sokolowski
daniel.sokolowski@unisg.ch
University of St. Gallen
Switzerland

Guido Salvaneschi
guido.salvaneschi@unisg.ch
University of St. Gallen
Switzerland

Abstract

Developers automate deployments with Programming Languages Infrastructure as Code (PL-IaC) by implementing IaC programs in popular languages like TypeScript and Python. Yet, systematic testing—well established for high-velocity software development—is rarely applied to IaC programs because IaC testing techniques are either slow or require extensive development effort. To solve this dilemma, we develop ProTI, a novel IaC unit testing approach, and implement it for Pulumi TypeScript. Our preliminary experiments with simple type-based test case generators and oracles show that ProTI can find bugs reliably in a short time, often without writing any additional testing code. ProTI's extensible plugin architecture allows combining, adopting, and experimenting with new approaches, opening the discussion about novel generators and oracles for efficient IaC testing.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; **Software functional properties**; *Orchestration languages*; • **Computer systems organization** → Cloud computing.

Keywords: Infrastructure as Code, DevOps, Testing, Fuzzing

ACM Reference Format:
David Spielmann, Daniel Sokolowski, and Guido Salvaneschi. 2023. Extensible Testing for Infrastructure as Code. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (SPLASH Companion '23).* ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Today IaC Testing is Slow or Effortful

Infrastructure as Code (IaC) [13] automates the complex management of resources and deployments for cloud applications. Beyond simplifying the orchestration, IaC introduces consistency and reproducibility into the deployment process [8, 10, 11, 15]. Today, Programming Languages IaC (PL-IaC) enables developers to tackle the complexity by implementing IaC programs in high-level programming languages like Python and TypeScript. The established PL-IaC solutions are Pulumi [14] and the Cloud Development Kits (CDKs) of AWS [1] and Terraform [9].

IaC programs describe the desired target state of the deployment as an append-only graph of immutable resource configurations and the dependencies between them. For instance, the Pulumi TypeScript program snippet in Figure 1 defines an AWS S3 bucket object by instantiation of an object of the resource's type and passing the resource's configuration to the constructor. Based on such a target state, the PL-IaC solution sends the *target configuration* of each resource to its deployment engine, which creates or updates the resources and returns the resources' *observed configuration* post deployment. The observed configuration is then accessible on the resources' object in the remainder of the IaC program.

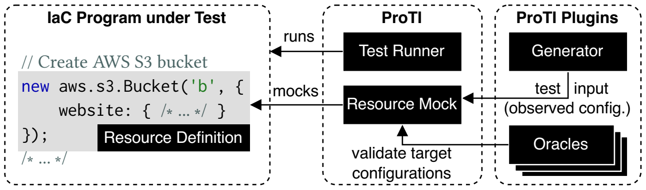
```
// Create AWS S3 bucket
new aws.s3.Bucket('b', {
    website: { /* ... */ }
});
/* ... */
```

Figure 1. High-level architecture of IaC testing with ProTI.

Using PL-IaC introduces the benefits of programming languages to IaC, but their challenges, too. Like traditional software, IaC programs are susceptible to bugs, which could prevent deployment, break functionality, or introduce security issues. Testing is a central technique to ensure program reliability. However, developers rarely test IaC programs systematically—as we found studying PL-IaC programs on GitHub. This stems from the practical infeasibility of current PL-IaC testing techniques. Integration testing is slow and resource-intensive, leading to increased costs and waiting times. Unit testing bypasses these issues at the expense of high manual development effort because each resource definition has to be mocked with logic to validate configurations and to generate artificial observed configurations, simulating the complex behavior of the cloud. In summary, developing reliable PL-IaC programs at high velocity is a challenge because current testing methods are either slow or demand significant development effort.

2 A Novel Approach for Efficient Testing

To solve this dilemma, we propose ProTI, a novel unit testing approach for PL-IaC programs we envisioned [16]. ProTI automatically mocks all resource definitions and employs a generator for test input and a set of oracles for validation (Figure 1). A test case is defined by the sequence of observed resource configurations the generator—and therefore the

SPLASH Companion '23, October 22–27, 2023, Cascais, Portugal
2023. ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

mocks—return. The oracles validate the target configuration of each resource, i.e., they check all mock inputs. ProTI runs the mocked IaC program quickly in many different configurations, halting once a bug is detected, or after a defined amount of runs or time in the successful case. To that end, ProTI is a property-based testing [2, 6] and fuzzing [17] tool for IaC programs. Both ProTI's generator and oracles are exchangeable plugins to empower reuse across IaC programs and be extensible towards existing and novel test case generation and validation techniques. In essence, ProTI enables fast and extensible unit testing *with* low development effort.

3 ProTI is Effective with Type-based Plugins

We implemented ProTI for Pulumi TypeScript based on the JavaScript testing framework Jest [5], the property-based testing library fast-check [4], and Pulumi's runtime mocking. The first ProTI plugins are type-based, using resource configuration types from Pulumi package schemas. The type-based generator provides test input by generating concrete values for observed resource configurations that comply with the resource's type. Similarly, the type-based oracle dynamically checks each concrete resource target configuration for compliance with the type information. As each Pulumi package defines a package schema, our plugins are out-of-the-box compatible with all Pulumi resources available today.

In a preliminary evaluation, we compare ProTI with current PL-IaC testing techniques on variants of a Pulumi TypeScript program, which deploys a website on AWS S3 displaying a word chosen randomly from a list. Three variants of the program have an error: (1) has an *error* that consistently causes the program to fail, (2) *async error* has an error in a callback that depends on the observed configuration of a resource, and (3) *off-by-one error* sometimes fails because it draws a random number that is one bigger than the highest index in the words list. Lastly, (4) is *correct*, and (5) *AWS RDS* additionally deploys a serverless database cluster. ProTI is equipped with the type-based generator and oracle plugins, and one execution runs 100 test cases. The alternative techniques to (a) ProTI for testing PL-IaC programs are (b) naïve *unit testing*, using a mock that neither validates nor generates configurations, (c) *dry running*, running the program without deploying it using Pulumi's preview, and (d) *integration testing*, simply executing the deployment.

Figure 2 shows the average run times over 12 repetitions and whether the error was (always) found. Dry running cannot find errors in code depending on observed configuration, and unit testing crashes on *async error*. ProTI exercises all code with realistic values in many different configurations, while *integration testing* only tests a single one. Thus, ProTI is the only testing technique that always detects all errors. ProTI is arguably fast, considering it runs up to 100 test cases in similar durations in which the alternatives run a single test case. Further, *correct* and *AWS RDS* show that ProTI takes

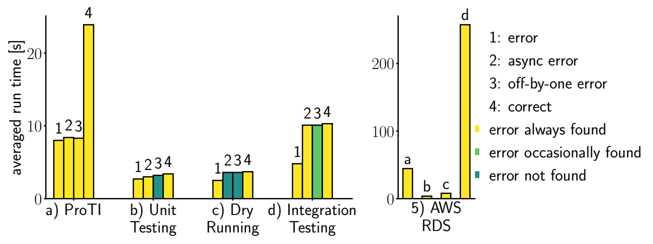

Figure 2. Average run time over 12 executions and whether an error was found for testing techniques on an IaC program.

longer with more resources (6 vs. 25) but is independent of their deployment duration. In contrast, integration testing is heavily slowed down by the long deployment time of the serverless database in *AWS RDS*. Lastly, no extra testing code was needed for ProTI, demonstrating the effort required for testing with ProTI.

4 Advanced Generators and Oracles

ProTI enables efficient testing of IaC programs, relieving developers from the tedious task to develop high-quality testing code. Instead, generator and oracle plugins that are implemented once can be reused to test many IaC programs. However, implementing effective plugins is not trivial. Ideal plugins would be *complete* and *correct*, i.e., an ideal generator would generate all possible configurations and no unrealistic ones, and an ideal oracle would detect all misconfiguration without identifying any valid configuration as incorrect (false positives). Further, ideal generators prioritize tests that are prone to fail, finding bugs as early as possible.

With the type-based plugins ProTI, showed already to be effective, even though these plugins are by far not ideal: The types are only an approximation (e.g., a valid TCP port must be an integer within a certain range, but its type 'number' includes also fractions and integers outside the valid range), and the test case prioritization is naïve. Yet, ProTI's effectiveness will further improve with more advanced plugins, and ProTI enables researchers and practitioners alike to easily explore existing and novel ideas, which could leverage, e.g., search-based [12], state-machine-based [3], or combinatorial-coverage-based [7] testing techniques. ProTI facilitates the discussion of novel test case generation and oracles, driving ever-more effective IaC testing.

Acknowledgments

This work has been co-funded by the Swiss National Science Foundation and by the University of St. Gallen.

References

[1] Amazon Web Services. 2023. AWS Cloud Development Kit. https://aws.amazon.com/cdk/ (Accessed: 2023-08-15).
[2] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the*

Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00). Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[3] Luis Eduardo Bueso de Barrio, Lars-Åke Fredlund, Ángel Herranz, Clara Benac Earle, and Julio Mariño. 2021. Makina: a new QuickCheck state machine library. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2021, Virtual Event, Korea, August 26, 2021*, Stavros Aronis and Annette Bieniusa (Eds.). ACM, 41–53. https://doi.org/10.1145/3471871.3472964

[4] Nicolas Dubien. 2022. fast-check: Property based testing framework for JavaScript/TypeScript. https://github.com/dubzzz/fast-check (Accessed: 2023-08-15).

[5] Facebook. 2023. Jest: Delightful JavaScript Testing. https://jestjs.io/ (Accessed: 2023-01-29).

[6] George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Softw. Eng. Notes* 22, 4 (1997), 74–80. https://doi.org/10.1145/263244.263267

[7] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover - Combining Combinatorial and Property-Based Testing. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 264–291. https://doi.org/10.1007/978-3-030-72019-3_10

[8] Michele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 580–589. https://doi.org/10.1109/ICSME.2019.00092

[9] HashiCorp. 2023. CDK for Terraform. https://developer.hashicorp.com/terraform/cdktf (Accessed: 2023-08-15).

[10] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. 2021. The Do's and Don'ts of Infrastructure Code: A Systematic Gray Literature Review. *Information and Software Technology* 137 (2021), 106593. https://doi.org/10.1016/j.infsof.2021.106593

[11] Leonardo A. F. Leite, Carla Rocha, Fabio Kon, Dejan S. Milojicic, and Paulo Meirelles. 2020. A Survey of DevOps Concepts and Challenges. *ACM Comput. Surv.* 52, 6 (2020), 127:1–127:35. https://doi.org/10.1145/3359981

[12] Andreas Löscher and Konstantinos Sagonas. 2018. Automating Targeted Property-Based Testing. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 70–80. https://doi.org/10.1109/ICST.2018.00017

[13] Kief Morris. 2021. *Infrastructure as Code: Dynamic Systems for the Cloud Age* (second ed.). O'Reilly Media, Inc., Sebastopol, CA, USA.

[14] Pulumi. 2022. Pulumi: Universal Infrastructure as Code. https://github.com/pulumi/pulumi (Accessed: 2023-08-15).

[15] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie A. Williams. 2019. A systematic mapping study of infrastructure as code research. *Inf. Softw. Technol.* 108 (2019), 65–77. https://doi.org/10.1016/j.infsof.2018.12.004

[16] Daniel Sokolowski and Guido Salvaneschi. 2023. Towards Reliable Infrastructure as Code. In *20th International Conference on Software Architecture, ICSA 2023 - Companion, L'Aquila, Italy, March 13-17, 2023*. IEEE, 318–321. https://doi.org/10.1109/ICSA-C57050.2023.00072

[17] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2021-10-26 21:30:20+08:00.