Unleashing the Giants: Enabling Advanced Testing for Infrastructure as Code

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

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

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

ABSTRACT
Infrastructure as Code (IaC) programs are written in imperative programming languages like Python or TypeScript while declaratively defining the target state of software deployments, which the IaC solution then sets up, e.g., Pulumi and AWS CDK. Through a repository mining study and analysis, we noticed that testing IaC programs poses a dilemma: current techniques are either slow and expensive or require prohibitively high development effort. To solve this issue, we introduce Automated Configuration Testing (ACT), enabling efficient testing with low development effort. ACT automates the tedious aspects of unit testing IaC programs and is extensible through a plugin system for test generators and oracles. ACT is already effective with simple type-based plugins, and leveraging existing giants, i.e., advanced test generation and oracle techniques, in new plugins will further boost its effectiveness.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging; Software functional properties; Orchestration languages.

KEYWORDS
Property-based Testing, Fuzzing, Infrastructure as Code, DevOps

 ACM Reference Format:

1 INTRODUCTION
Infrastructure provisioning and application deployment are increasingly complex tasks that require automation to ensure organizations can adopt their applications quickly and frequently to changing requirements. Infrastructure as Code (IaC) [15] is the DevOps technology to automate deployments. With state-of-the-art declarative IaC, developers only describe the target state of their deployment, and the IaC solution achieves it by comparing the current infrastructure with the target state, deriving the required actions.

2 THE DILEMMA OF IAC PROGRAM TESTING
The reliability of IaC programs is vital because faulty deployments can cause entire systems to malfunction and severe security vulnerabilities. As IaC programs use widely adopted programming languages with mature ecosystems, a whole array of existing quality assurance techniques is directly applicable, especially for testing. However, in a previous study on public IaC programs on GitHub, we found that only 25% implement tests, dropping to 1% for Pulumi [21]. We focus on Pulumi because it is the most expressive solution for IaC programs. It is the only one allowing computing on output configuration of just deployed resources in the IaC program—a powerful feature that, unfortunately, impedes testing.

To find out why developers do not write tests, we analyzed IaC program testing techniques and identified a dilemma: integration testing is very slow and resource-intensive for IaC programs. The only reliable alternative is unit testing, which is quick. However, due to the declarative nature of IaC programs, developing effective unit tests requires tremendous effort. Developers must (1) mock all resource definitions, often most of the IaC program code. The mocks must (2) validate the input configuration for each defined resource, making the mocks test oracles. Further, they have to (3) provide post-deployment output configuration for each defined resource, which is accessible in the remaining IaC program execution and, thus, test input, making the mocks also test generators. Implementing both good oracles and generators is tedious and replicates the logic of the IaC program and the infrastructure, causing a lot of tightly coupled testing code and slowing down future changes.

3 AUTOMATED CONFIGURATION TESTING
To solve the dilemma of IaC program testing, we envisioned Automated Configuration Testing (ACT) [19] inspired by property-based testing [3] and fuzzing [23] techniques. ACT automates the tedious work of implementing unit tests by automatically mocking all resource definitions. ACT leverages a plugin interface for the involved aspects of the mocks, i.e., test generation and oracles, allowing the reuse and exchange of various techniques (Figure 1). At this high degree of automation, ACT efficiently tests the IaC program under test.
by running it automatically in many configurations quickly. ACT sends every resource input configuration to the generator plugin, providing the resources’ output configuration to use in the rest of the execution, and to all oracle plugins, verifying them. For generalized properties, developers can reuse plugins centrally maintained by the community, enabling IaC program testing without writing additional testing code. To additionally test for application-specific properties, developers can utilize type-based ACT plugins leveraging resource types from Pulumi type packages. Even though these plugins are simplistic and not very precise, our evaluation with thousands of Pulumi TypeScript programs from GitHub and artificial benchmarks shows that ACT is effective. A single ProTI test run typically takes hundreds of milliseconds, allowing testing IaC programs in hundreds to thousands of configurations quickly, effectively finding bugs—even in corner cases.

4 STANDING ON THE SHOULDERS OF GIANTS FOR ADVANCED IAC PROGRAM TESTING

ACT’s effectiveness depends on the generators and oracles. While our simple type-based plugins are already useful, advanced techniques promise a significant boost for the whole approach. At this point, existing and novel advanced automated testing techniques can be integrated into ProTI (ACT), effectively preventing malfunctioning and insecure deployments. In initial experiments, we already demonstrated that such integration is suitable and simple, where we implemented slim wrapper plugins integrating the Daikon invariant detector [6] and the Radamsa fuzzer [10]. We now outline ideas and approaches for future explorations.

So far, the test generation we have used is purely random and uninformed of the program and previous test runs. For better test input generation strategies, the automated testing literature proposed various approaches, including techniques leveraging test coverage and feedback information [8, 12, 16] as well as search- and grammar-based techniques [13, 22].

The problem of finding good test oracles is not limited to IaC program testing, either. Our current type-based oracles are imprecise and cannot cover validation across properties or take other contexts into account. However, there are oracle strategies that should be explored in this domain. Promising directions are finding IaC properties for differential [7], metamorphic [2], intramorphic [18], and learning-based testing approaches [4, 11].

Figure 1: ACT components and their interaction.

ACKNOWLEDGMENTS

This work has been co-funded by the Swiss National Science Foundation (SNSF, No. 200429).

REFERENCES