Generating Tests that Cover Input Structure

Nataniel Pereira Borges Jr.[1] Nikolas Havrikov[2] Andreas Zeller[3]

Abstract: To systematically test a program, one needs good *inputs*—inputs that are *valid* such that they are not rejected by the program, and inputs that *cover* as much of the input space as possible in order to reach a maximum of functionality.

We present *recent techniques to systematically cover input structure.* Our k-path algorithm for grammar production [HZ19] systematically covers syntactic elements of the input as well as their combinations. We show how to *learn* such input structures from graphical user interfaces, notably their *interaction language* [DBJZ19]. Finally, we demonstrate that knowledge bases such as DBPedia can be a reliable source of semantically coherent inputs [Wa20]. All these techniques result in a significantly higher code coverage than state of the art.

Keywords: grammar; coverage; automated testing; input generation; knowledge-base; android

1 Achieving Grammar Coverage

Testing programs with randomly generated inputs is a cost-effective means to test programs for robustness. However, to reach deep layers of a program, the inputs must be syntactically valid. Using a grammar to specify the language of program inputs lends itself well to solving this problem: A grammar-based test generator uses such a grammar to expand its start symbol into further symbols repeatedly until only terminal symbols are left, constituting an input. When generating inputs, intuitively, a high variation in the inputs should lead to a high variation in program behavior.

We present a notion of grammar coverage called k-path coverage and an approach for quickly achieving it. A k-path consists of k consecutive symbols along a valid derivation sequence in a derivation tree or a grammar. For any given grammar the number of k-paths is finite and one can generate a set of inputs exhibiting all of them by greedily deriving towards yet unvisited k-paths while keeping track of any k-paths covered incidentally. Having fully derived a targeted k-path, we promptly close off the current tree as quickly as possible and start generating a new one for the next unvisited k-path.

[1] CISPA Helmholtz Center for Information Security, Saarbrücken nataniel.borges@cispa.de
[2] CISPA Helmholtz Center for Information Security, Saarbrücken nikolas.havrikov@cispa.de
[3] CISPA Helmholtz Center for Information Security, Saarbrücken zeller@cispa.de

doi:10.18420/SE2021_31

2 UI Element Interactions

In the context of testing a mobile app, automated test generators systematically identify and interact with its user interface elements. One key challenge hereby is to synthesize inputs that effectively and efficiently cover app behavior. This is usually approached by having a model mapping UI elements to actions they usually accept. Such a model can be mined statically from an app or dynamically from observing its executions. Both these approaches, however, are biased towards the distribution originally mined. They work well if the app under test is similar to those used to train the model, but fail if it is dissimilar.

We present a technique that automatically adapts the model to the app at hand by approaching test generation as an instance of the multi-armed bandit problem, where a finite set of resources (actions) has to be distributed among competing alternatives (UI elements) to increase its reward (test quality). We use reinforcement learning to address test generation from this perspective and to systematically and gradually adjust our test generation strategy towards the application under test.

3 Using Knowledge Bases

Staying in the context of mobile apps, many take complex data as input, such as travel booking, map locations, or online banking information. These inputs are, however, expensive to generate manually and challenging to synthesize automatically. Past research indicated that knowledge bases could be a reliable source of semantically coherent inputs.

We propose an approach for leveraging knowledge bases for mobile app test generation comprising four steps: Given a UI state, we start by identifying and matching descriptive labels with input fields according to a set of metrics based on the Gestalt principles. We then use natural language processing to extract a *concept* associated with each label. We use the extracted concepts, instead of the original labels, to query knowledge bases for input values. Finally, we fill all input elements with the queried values and randomly interact with the non-input elements.

Bibliography

[DBJZ19] Degott, Christian; Borges Jr., Nataniel Pereira; Zeller, Andreas: Learning User Interface Element Interactions. In: ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019). July 2019.

[HZ19] Havrikov, Nikolas; Zeller, Andreas: Systematically Covering Input Structure. In: IEEE/ACM International Conference on Automated Software Engineering (ASE 2019). November 2019.

[Wa20] Wanwarang, Tanapuch; Borges Jr., Nataniel Pereira; Bettscheider, Leon; Zeller, Andreas: Testing Apps With Real-World Inputs. In: 1st IEEE/ACM International Conference on Automation of Software Test (AST 2020). May 2020.