One-Size-Fits-None? Improving Test Generation Using Context-Optimized Fitness Functions

Gregory Gay
Department of Computer Science & Engineering
University of South Carolina
Columbia, SC, USA
greg@greggay.com

***Abstract*—Current approaches to search-based test case generation have yielded limited results in terms of human-competitiveness. However, effective search-based test generation relies on the selection of the correct fitness functions—feedback mechanisms—for a chosen goal. We propose that the key to overcoming these limitations lies in infusing domain knowledge and context into the fitness functions used to guide the search and the ability to automatically optimize the fitness functions used when generating tests for a given class, goal, and algorithm.**

***Index Terms*—Automated Test Generation, Search-Based Software Testing, Search-Based Software Engineering**

I. Introduction

Test input generation can naturally be seen as a search problem. Testers approach input selection with a **goal** in mind—causing the program to crash, maximizing code coverage, or any number of other potential goals. Of the near-infinite choices of input that could be provided, the tester seeks specific selections that meet the chosen goal. Given this goal, an optimization **algorithm** can systematically sample the space of test input, guided by feedback from one or more **fitness functions**—scoring functions that judge the optimality of the chosen input. In other words: $algorithm + fitness\ functions \implies goal$.

Effective search-based test generation relies on the selection of the correct sampling mechanism—the right algorithm—and, perhaps more importantly, the right feedback mechanism—the right fitness functions. Fitness functions offer the algorithm the *feedback* and *guidance* needed to locate better solutions, making their design and use important. A great deal of attention has been paid to the algorithm variable in the test generation equation [10]. Comparatively less attention has been paid to the fitness functions [2]. We propose that the key to moving search-based test generation into "production"—advancing the field to the point where we produce human-competitive tests—requires a deep look at the other variables of the equation, particularly the fitness functions.

II. The Limitations of General Approaches

The default goal of testing is to find faults in class or system-under-test (CUT, SUT). Even if test generation is performed to satisfy a measurable goal, the ultimate goal of the tester is to usually to find issues in the SUT. In that regard, search-based test generation still has great room for improvement [2].

Arguably, the majority of approaches to search-based generation are based on code coverage, often specifically based on Branch Coverage[1]. Branch Coverage—and many similar criteria–measure *how much* of the code has been executed, with the rationale being that faults cannot be detected without execution [5]. This rationale is intuitive, coverage can be measured efficiently, and there are fitness function representations that can quickly improve coverage [10].

However, coverage has a tenuous relationship with fault detection [6]. *How* code is executed is far more important than whether it was executed. Coverage criteria are able to drive execution, but tend to impose few constraints on how that code is executed. Even for non-coverage fitness functions, there tend to be a large number of input choices that satisfy the imposed constraints—only a small subset of those inputs are actually useful for inducing failure.

Each SUT—even each class in a SUT—is unique. By focusing on the idea of universal applicability, we may have unwittingly limited the efficacy of automated test generation. General "*one-size-fits-all*" criteria do not reflect how human testers behave. A human tester may measure and take advice from Branch Coverage, but rarely do they base their input selection on what will quickly increase coverage. Human testers are *driven by context*—informed by the requirements of the SUT, product domain, and past experience.

A narrow focus on generalized criteria explicitly lacks context. We hypothesize that this limits the efficacy of automated generation. Test generation—specifically, the fitness functions used to guide generation, must evolve to incorporate the context that humans use in developing test cases. *We must generate test cases in a more human-like manner*—based on the unique attributes of the CUT or SUT.

III. Knowledge-Infused Fitness Functions

Fitness functions offer equations encoding a set of knowledge and constraints. Contextual knowledge can be encoded into these equations. Two key research challenges are to identify the *types* of knowledge that should be incorporated and the ideal *means* of incorporation.

Support offered by National Science Foundation grant CCF-1657299.

[1]For each statement that can cause execution to diverge—i.e., `if` and `case`—test input should ensure that at all outcomes are executed [5].

Context-based fitness functions can be co-optimized with traditional coverage-based functions, taking advantage of their ability to rapidly explore the code structure. Branch Coverage offers a powerful feedback mechanism, granting the ability to reach relevant portions of code. For example, tests generated to simultaneously maximize Branch Coverage and an exception count are often more effective than test suites generated using either function alone [3]. The combination is able to take advantage of the feedback offered by Branch Coverage and the input selection bias induced by the exception count.

In that vein, lightweight contextual fitness functions could produce effective test suites. These functions could be based on targeted aspects of a SUT or CUT such as:

Product Domain: Mobile battery/data consumption.
Testing Scenarios: Integration prioritization.
Code Attributes: Private code, Boolean masking.
Non-Functional Priorities: Readability, memory usage.

An open research challenge is in discovering a process for easily formulating new contextual fitness functions.

A more heavyweight, but promising contextual approach is based around **system requirements**—constraints on the behavior of the SUT, usually written as textual statements. This text can be distilled into Boolean predicates, which could be incorporated into fitness functions in a way similar to code coverage. Requirements-based coverage criteria, like Unique-First-Cause (UFC) Coverage [11], transform predicates into coverage goals. These goals—path-based constraints over program variables—can be translated into numeric cost functions like those used for fitness functions based on Branch Coverage.

There are challenges associated with the use of requirements in this manner. Often, requirements require the generation of a series of actions, rather than a single action—adding difficulty to the search process [1]. Further, translating natural language into usable properties may require significant effort, while code coverage can be completely automated. Recent advances in natural language processing are promising in this regard, including recent work that extracts properties from code documentation [4]. Such techniques could potentially be used to automatically inform a fitness function.

IV. Automated Tuning of Fitness Functions

Implementing new context-infused fitness could help us achieve a wider variety of testing goals. However, a question remains—in fact, it becomes more imperative—*which* fitness functions should we employ?

This question lacks an obvious answer. Given our hypothesis that all classes are somewhat unique, different combinations of one or more fitness functions may best satisfy a given goal for different systems. It follows that fitness function selection itself offers a secondary optimization problem. Given a CUT or SUT, a goal, and an algorithm, which fitness functions should be employed to best reach the chosen goal?

Hyperheuristic search algorithms automatically tune aspects of their approach to better solve the current problem of interest [8]. We propose that hyperheuristic search could be used to optimize the set of applied fitness functions. Using optimization, artificial intelligence, or machine learning techniques, we could *learn* the ideal fitness functions for our combination of CUT, algorithm and goal.

How to perform this process is an open question, but there are promising avenues of exploration. Given a machine-measurable goal, reinforcement learning techniques can identify fitness functions ideal for improving attainment of that goal [8]. This could be a reasonable way to choose an ideal subset of fitness functions for a CUT. Transfer learning techniques could be employed to observe and extract patterns from the results of this optimization process, and build models that can be applied to new classes or systems [7].

Rather than crafting a complex—and needlessly difficult to optimize—set of functions, we also must consider ways to directly tune and evolve a single fitness function suited to the CUT, algorithm, and goal. We could directly craft and evolve customized fitness functions consisting of goals relevant to the CUT or SUT—combining elements of structural coverage, execution properties, problem domain, and system requirements as appropriate. A promising direction in this regard has been proposed by McMinn et al., who suggest the formulation of fitness functions as tree structures, akin to those used in Genetic Programming [9]. Such a formulation could be directly optimized by a hyperheuristic search.

V. Conclusions

We propose that overcoming limitations in the human-competitiveness of search-based test generation requires infusing contextual knowledge into fitness functions and automatically tuning these functions based on the SUT or CUT, goal, and algorithm. This agenda presents a number of research challenges to overcome—how and what type of knowledge to formulate, how to tune functions, and how to match functions to new systems and goals. However, we hypothesize that overcoming these challenges will yield great benefit.

References

[1] G. Gay. Challenges in using search-based test generation to identify real faults in mockito. In *Proc. of SSBSE 2016*.

[2] G. Gay. The fitness function for the job: Search-based generation of test suites that detect real faults. In *Proc. of ICST 2017*.

[3] G. Gay. Generating effective test suites by combining coverage criteria. In *Proc. of SSBSE 2017*.

[4] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic generation of oracles for exceptional behaviors. In *Proc. of ISSTA 2016*.

[5] A. Groce, M. A. Alipour, and R. Gopinath. Coverage and its discontents. In *Proc. of Onward! 2014*.

[6] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. of ICSE 2014*.

[7] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund. Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. In *Proc. of ESEC/FSE 2018*.

[8] Y. Jia. Hyperheuristic search for sbst. In *Proc. of SBST '15*.

[9] P. McMinn, M. Harman, G. Fraser, and G. M. Kapfhammer. Automated search for good coverage criteria: Moving from code coverage to fault coverage through search-based software engineering. In *Proc. of SBST 2016*.

[10] A. Panichella, F. M. Kifetew, and P. Tonella. A large scale empirical comparison of state-of-the-art search-based test case generators. *IST*, 104:236 – 256, 2018.

[11] M. Whalen, A. Rajan, and M. Heimdahl. Coverage metrics for requirements-based testing. In *Proc. of ISSTA 2006*.