Verification++: Automated Test Generation Using Type Theory

Ali Al Akbar Haidoura
American University of Beirut
ahh68@mail.aub.edu

1 Introduction

Software testing is an important phase of the software engineering cycle, leading to the early discovery of bugs and problems of varying danger. However, choosing a convenient coverage metric and designing the test cases accordingly can be a troublesome task. The goal of this project is to create a tool that analyses C++ source code and automatically generates test cases leading to complete code coverage modulo type theory.

2 Methodology

The tool statically analyses the source code to determine the various variables and data structures used in it, along with their types and the operations using them. Then, first order logic predicates (we call them boolean atoms) are formed describing different relations between these variables, which are then combined (using negations and conjuctions) to derive formulae which define different variable classes which are equivalent modulo the execution of the program. Next, an instrumented version of the code is run against satisfying assignments of the different equivalence classes, where the result of each run is used to generalize the classes and prune any non necessary branches of the decision diagram constructed after the generated formulae. In the end, we would have a set of equivalence classes which cover all of the program space, each defining a different execution path. We use satisfying assignments of these classes as test cases which should result in full coverage testing.

3 Implementation

For the static analysis part, we make use of Clang's AST to determine the variables and operations in the code. We use the operations, specifically relational operations to infer the type theory (as defined in [1]) of the code, the atoms of which are combined to make formulae that will be passed to the Z3 SMT solver to check for satisfiability, where a satisfiable formula represents a space of values for the variables of the program, where any two assignments satisfying the same formula should produce the same execution path. In parallel, the code gets instrumented by replacing the variables with objects that support the same operations and additionally log instrumentation information. The instrumented code is then run against the satisfying assignments from the equivalence classes, and we use the logged information to infer which formuas produce the same paths and which don't affect the execution, which helps in further pruning and optimizing decision diagram constructed using the formuas. In the end, the remaining classes are the classes that produce different behaviors and therefore which should be tested for, and testing for these classes should produce a complete coverage of all program behaviors.

4 Completed Work

So far, the static analysis part of the source code is functional. For the formula generation, we will be using the tool descibed in [1] as an inspiration, and might use most of the core functionality of SpecConstruct and integrate it in this tool. As for the instrumentation part, a few attempts have been made for writing the instrumenting classes, but that requires further work. We still need to integrate all the parts together and make sure they interact properly (especially the formula generation and code instrumentation parts).

References

[1] Paul C. Attie, Fadi Zaraket, Mohammad Noureddine, and Farah El-Hariri. Specification construction using smt solvers. 2013.