To Run What No One Has Run Before: Executing an Intermediate Verification Language*

Nadia Polikarpova, Carlo A. Furia, and Scott West

Chair of Software Engineering, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. When program verification fails, it is often hard to understand what went wrong in the absence of concrete executions that expose parts of the implementation or specification responsible for the failure. Automatic generation of such tests would require "executing" the complex specifications typically used for verification (with unbounded quantification and other expressive constructs), something beyond the capabilities of standard testing tools.

This paper presents a technique to automatically generate executions of programs annotated with complex specifications, and its implementation for the Boogie intermediate verification language. Our approach combines symbolic execution and SMT constraint solving to generate small tests that are easy to read and understand. The evaluation on several program verification examples demonstrates that our test case generation technique can help understand failed verification attempts in conditions where traditional testing is not applicable, thus making formal verification techniques easier to use in practice.

1 Help Needed to Understand Verification

Static program verification has made tremendous progress, and is now being applied to real programs [16,11] well beyond the scale of "toy" examples. These achievements are impressive, but still require massive efforts and highly-trained experts. One of the biggest remaining obstacles is understanding failed verification attempts [19]. Most difficulties in this area stem from inherent limits of static verification, and hence could benefit from complementary *dynamic* techniques.

Static program proving techniques—implemented in tools such as Boogie [17], Dafny [18], and VeriFast [8]—are necessarily incomplete, since they target undecidable problems. Incompleteness implies that program verifiers are "best effort": when they fail, it is no conclusive evidence of error. It may as well be that the specification is sound but insufficient to prove the implementation correct; for example, a loop invariant may be too weak to establish the postcondition. Even leaving the issue of incomplete specifications aside, the feedback provided by failed verification attempts is often of little use to understand the ultimate source of failure. A typical error message states that some executions might violate a certain assertion but, without concrete input values that trigger the violation, it is difficult to understand which parts of the programs

* Work partially supported by SNF grants 200021-137931 (FullContracts), 200020-134974 (LSAT), and 200021-134976 (ASII); and by ERC grant 291389 (CME).

should be adjusted. And even when verification is successful, it would still be useful to have "sanity checks" in the form of concrete executions, to increase confidence that the written specification is not only consistent but sufficiently detailed to capture the intended program behavior.

Dynamic verification techniques are natural candidates to address these shortcomings of static program proving, since they can provide concrete executions that conclusively show errors and help narrow down probable causes. Traditional dynamic techniques based on *testing* are, however, poor matches to the capabilities of static provers. Testing typically targets simple properties, such as out-of-bound and null dereferencing errors, or, only in a minority of cases, lightweight executable specifications (e.g., contracts). Program provers, in contrast, work with very expressive specification and implementation languages supporting features such as nondeterminism, unbounded quantification, infinitary structures (sets, sequences, etc.), and complex first- or even higher-order axioms; none of these is executable in the traditional sense. As we argue in Sec. 2, however, even relatively simple programs may require such complex specifications. Program provers also support modular verification, where sufficiently detailed specifications of modules or routines are used in lieu of missing or incomplete implementations; this is another scenario where runtime techniques fall short because they require complete implementations.

In this paper, we propose a technique to generate executions of programs annotated with complex specifications using features commonly supported by program provers (nondeterminism, unbounded quantification, partial implementations, etc.). The technique combines symbolic execution with SMT constraint solving to generate small and readable test cases that expose errors (failing executions) or validate specifications (passing executions).

The proposed approach supports executing both imperative and declarative program elements, which accommodates the *implementation* semantics of loops and procedure calls, defined by their bodies, as well as their *specification* semantics, used in modular verification, where the effect of a procedure call is defined solely the procedure's pre- and postcondition and the effect of a loop by its invariant. The implementation semantics is useful to discriminate between inconsistent and incomplete specifications; while the specification semantics makes it possible to generate executions in the presence of partial implementations, as well as to expose spurious executions permitted by incomplete specifications.

Our technique simplifies the constraints passed to the SMT solver, only targeting the values required for a particular symbolic execution. This avoids the solver getting bogged down when reasoning about complex specifications—a problem often arising with program provers—without need for additional guidance in the form of quantifier instantiation heuristics. The simplification also improves the predictability of test case generation. Combined with model minimization techniques, it produces short—often minimal-length—executions that are quite easy to read. While constraint simplification might also produce false positives (infeasible executions), the evaluation of Sec. 5 shows that this rarely happens in practice: the small risk amply pays off by producing easy-to-understand executions, symptomatic of the rough patches in the implementation