A Coq Framework For Verified Property-Based Testing

(Extended Abstract)

Zoe Paraskevopoulou[1,2,3] Cătălin Hriţcu[3] Maxime Dénès[3]
Leonidas Lampropoulos[4] Benjamin C. Pierce[4]

[1]ENS Cachan [2]NTU Athens [3]Inria Paris-Rocquencourt [4]University of Pennsylvania

Abstract

We introduce a novel methodology for formally verified property-based testing and implement it as a framework on top of the QuickChick testing plugin for Coq.[1] Our verification framework is aimed at proving the correctness of executable testing code with respect to a high-level specification, which captures the conjecture under test in a more direct way. To this end, we provide a systematic way for reasoning about the set of outcomes a random data generator can produce with non-zero probability. We have used our methodology to prove the correctness of most QuickChick combinators, with respect to the axiomatic semantics of a small number of primitive ones. We have also applied our methodology on a red-black tree example and made good progress on a more complex non-interference one. These encouraging preliminary results indicate that this verification methodology is modular, scalable, and requires minimal changes to existing code.

Extended Abstract

While property-based testing is often an effective way for quickly finding bugs and increasing software quality, testing errors can conceal important bugs and thus reduce its benefits. In this work [5] we introduce a novel methodology for formally verified property-based testing (Figure 1); this methodology is embodied by a framework built on top of the QuickChick [3] testing plugin for Coq, which provides similar functionality to that of Haskell's QuickCheck [2].

Similar to QuickCheck, in QuickChick one can write efficiently executable programs, referred to as checkers, that express a conjecture about the system under test. QuickChick then tests the validity of the conjectures by running the checkers on a large number of randomly generated inputs. Some data generators are built-in, but the user can write custom generators using a library of generator combinators. However, one may wonder, how much confidence can we have about the program under test adhering its specifications, when testing cannot find any more bugs? In fact there are quite a few things that can tamper with the effectiveness of property-based testing. A generator may fail to cover the whole input space or may only generate only data that fails to satisfy a precondition and thus for which the property holds vacuously. Checkers may fail to capture the desired high-level specification, especially when it comes to large systems with complex invariants.

In this work [5] we propose a new way to gain formal guarantees about the quality of testing by showing that the conjecture under test corresponds to a high-level declarative specification. This is enabled by the fact that all testing code written by the user and the large majority of QuickChick is written in Coq itself. To make verification convenient, we devise a mechanism to automatically map both generators and checkers to declarative semantics. The semantics of each checker is a logical proposition (Coq sort `Prop`) that we then prove equivalent to the high-level declarative specification that we claim is being tested. The guarantee that this method provides is that if we could enumerate the output space of the used generators without producing any counter-examples then we would have a proof by exhaustion for the desired declarative specification. While exhaustion is very rarely possible in practice and would be incompatible with our randomized testing approach, this theoretical guarantee ensures that we are testing the correct conjecture. Most bugs in real generators and checkers are breaking even this rather weak guarantee.

Generators We reason about each generator in terms of its set of outcomes, the set of values that have non-zero probability of being generated (in Coq we represent this as an `Ensemble`, a function from α to `Prop`). We map most generators in QuickChick and all user generators to sets of outcomes by giving an axiomatic semantics to a small set of primitive generator combinators (e.g., `return` and `bind`). Using the set of outcomes semantics we were able to prove that the non-primitive combinators provided by QuickChick are correct (i.e. sound and complete) with respect to high-level predicates that capture their expected behavior in an intuitive way. For instance the `listOf` combinator, which given a generator G generates lists of elements produced by G, has the following specification: $[\![\texttt{listOf}\ G]\!] \equiv \{xs \mid \forall x \in xs, [\![G]\!]\ x\}$ (where $\equiv$ denotes extensional `Ensemble` equivalence).

Checkers Using the set of outcomes semantics for generators we can map checkers to propositions that capture the conjecture that they test. Checkers are represented in-

[1]`https://github.com/QuickChick`

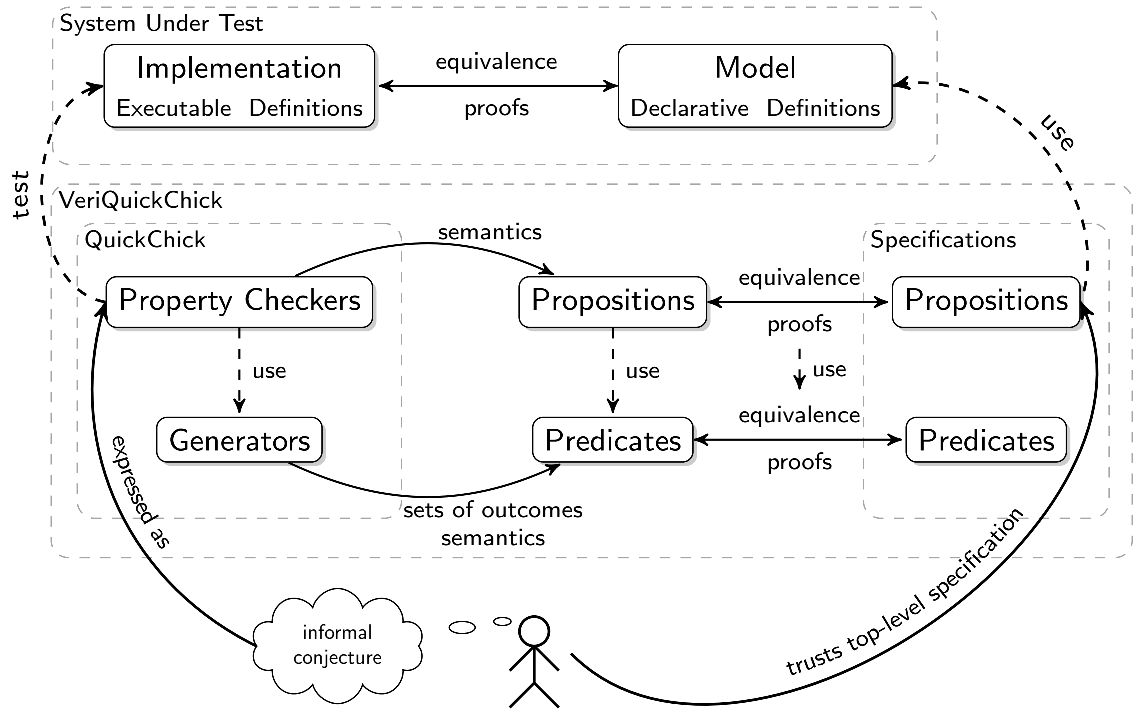

Figure 1: The proposed verification methodology

ternally as generators of test results, so they are also probabilistic programs. This allows us to map each checker to a set of outcomes and consequently to a logical proposition by requesting all elements of its set of outcomes to be successful testing results. Similarly to generators, we use this semantics to prove correctness for checker combinators by showing logical equivalence between the proposition automatically derived from the semantics and a proposition expressing the desired high-level specification. One interesting example is the `forAll` combinator that given a generator G and a Boolean predicate P tests whether P holds for the outputs of G. The semantics of this combinator is a proposition of the form

$$\forall x \in [\![G]\!], P\ x = \mathit{true}$$

Experiments Beyond verifying a large part of QuickChick itself, we applied our verification methodology on a simple red-black trees example and on a complex already existing infrastructure for testing noninterference [4]. In all cases this required minimal changes to the existing code. Our framework encourages the user to structure the proofs in a compositional way, achieving modularity and thus more robust and scalable proofs. Generally, the proofs closely follow the structure of the code. The proofs for derived generators and checkers only need to use the specifications of the combinators they use, but they are independent of their concrete implementations.

Future Work Verification in our framework is at the moment a manual process, still we believe the sets of outcomes abstraction is highly suitable for automatic verification, and we will explore that in the future. Beyond finishing the noninterference case study we also plan to apply our verification framework to testing other security monitors [1]. We will also exploit automation in the verification of the QuickChick combinators and reduce the number of primitives that are given an axiomatic semantics to the absolute minimum, turning QuickChick into the first formally verified PBT framework. More ambitiously, we would like to repeat this verification while taking into account probabilities. Finally, we are working on a language-based framework for producing property-based generators, and we would like to use our verification methodology to prove the correctness of that framework.

References

[1] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: A framework for verified, hardware-assisted security monitors. Draft, July, 2014.

[2] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2000.

[3] M. Dénès, C. Hriţcu, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. QuickChick: Property-based testing for Coq. The Coq Workshop, 2014.

[4] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2013.

[5] Z. Paraskevopoulou and C. Hriţcu. A Coq framework for verified property-based testing. Internship Report, Inria Paris-Rocquencourt, 2014.