Dependable Property-Based Testing

Advisor: Cătălin Hriţcu ⟨catalin.hritcu@gmail.com⟩

Instiition: INRIA Paris, Prosecco Team

Location: 2 rue Simone Iff, 75012 Paris, France

Language: English

Existing skills or strong desire to learn:

- functional programming (e.g. ML or Haskell);
- property-based testing (e.g. QuickCheck);
- optional: formal verification in Coq or F*;
- optional: logic programming, constraint solving, probabilities, algorithmic optimization, co-induction

Introduction

Context Property-based testing (PBT) and formal verification are two complementary techniques for reducing the prevalence of software errors. PBT is a systematic technique that promises to both reduce the cost and increase the thoroughness of testing. It allows software testers to write executable specifications capturing expected properties of the system under test, or a small part of it, and it semi-automatically tests these properties on large sets of test data, e.g., randomly generated according to some probability distribution. When a counterexample is found it is automatically shrunk to a minimal one, which is displayed to the user. QuickCheck [6], the first popular PBT tool, targets Haskell and is widely used in the functional programming languages community. QuickCheck has been ported to all mainstream programming languages and has spurred significant industrial interest.

While PBT is effective at finding errors, no amount of testing is enough to guarantee the absence of errors. Only formal verification can provide the strong guarantees required for safety- and security-critical software, for which errors can have disastrous real-world consequences. Continuous progress in formal verification in the past decades has culminated with many recent milestones (e.g., seL4, CompCert, RockSalt, CertiKOS, miTLS) which show beyond any doubt that (with enough effort) formal verification can scale up to sizable software artifacts, such as compilers, operating system kernels, and cryptographic protocols.

Despite this great progress, formal verification in a proof assistant is still prohibitively expensive for mainstream adoption. Carrying out a formal proof while designing even a relatively simple system can be an exercise in frustration, with a great deal of time spent attempting to prove things about broken definitions, and countless iterations for discovering the correct lemmas and strengthening inductive invariants.

Project's Goal The goal of this project is *to achieve the lower cost of PBT and the stronger formal guarantees of verification in one single system.* For this we will tightly integrate PBT and formal verification in the Coq proof assistant and achieve a virtuous cycle in which testing helps verification and verification helps testing. Testing will decrease the cost of verification by allowing users to find errors in definitions and conjectured properties early in the design process, and to postpone verification attempts until they are reasonably confident that their system is correct. Easily understandable counterexamples will guide users not just for producing correct system designs, but also for constructing the formal evidence of their correctness during the verification process, by quickly validating proof goals, potential lemmas, and inductive invariants. All this will dramatically decrease the number of costly failed proof attempts in Coq developments, which will lower the barrier to entry and increase adoption of the Coq proof assistant. Moreover, integrating PBT with Coq will provide an easier path going from systematic testing to formal verification, by encouraging developers to write specifications that can be only tested at first and later formally verified. Finally, our solution will be fully general, and thus also be useful to regular software developers for testing outside proof assistants.

Since in practice dependable PBT can itself be prohibitively difficult, the main objective of this proposal is to reduce the cost of PBT, while maintaining its full generality and programmability. We will achieve this by a combination of programming language design, code synthesis, and algorithmic optimization. In particular we will introduce Luck, a novel domain-specific language for property-based generators that leads to an order of magnitude reduction in code size using ideas from functional logic programming and constraint solving, provides automation of common patterns, yet is fully customizable, and keeps the user fully in control.

Verification will help testing by providing stronger formal foundations for what is otherwise a mostly empirical discipline, going from "oh well, it's just testing" and experience- and intuition-driven design patterns to a rigorous science of testing. In particular, the integration of PBT in Coq enables us to prove properties about testing itself and to evaluate testing quality more systematically. For the former, we will build both an automated verification tool for testing code, relying on a recently introduced *possibilistic abstraction* for generators [11], and a more precise interactive framework for probabilistic verification. For the latter, we will build a polarized mutation testing framework that uses the declarative Coq definitions to systematically change the artifact under test to introduce all pointwise bugs from an interesting class. Our initial experiments with all these ideas are promising, but significant research challenges remain to be solved and our solutions will lead to improvements to the state of the art both in testing and in formal verification. We will measure success by performing several realistic case studies targeting widely-used TLS implementations, secure compilers, tag-based reference monitors, and complex type-checkers, including F* [12] and the Coq type-checker itself.

Related work While we are among the few [7] to investigate how proving can help testing [11], the potential of PBT reducing the cost of proving was already enough to lead to its integration into proof assistants such as Isabelle [4] and ACL2 [5]. Inspired by this, we have recently ported the QuickCheck framework to Coq, resulting in a prototype Coq plugin called QuickChick.[1] While these previous efforts have had some success, they are still very far from achieving the full potential of this integration. Most previous tools are either aimed exclusively at fully automated testing, which makes them extremely limited in what they can test (and even more limited in what they can test *well*) or they provide a powerful set of primitive operators, usually inspired from QuickCheck, which allows the user to write all the necessary testing code by hand; doing this well can be so costly and error prone that it often negates most benefits of testing.

Challenges and Scientific Objectives

The rest of document is aimed at briefly presenting various research directions to which a student or young researcher could contribute (during a research internship, MSc or PhD thesis, PostDoc, etc). The list is not exhaustive, and if you are interested you should get in contact and we will together try to find a topic that is in sync with your interests and expertise.

Challenge 1: Dependable testing requires significant effort Currently, a QuickChick user has to write efficiently executable variants of the proof-oriented artifacts she uses for verification, this includes both the system (e.g. a type system, a dynamic monitor) and the properties under test (e.g. progress, preservation, noninterference). We call the executable variant of the property under test a *checker* for that property. Checkers are, however, not sufficient for testing conditional properties with sparse pre-conditions; for instance generating random lists and then filtering out the ones that are not sorted leads to extremely inefficient testing. In such common cases the user has to additionally provide *property-based generators* that efficiently produce only data satisfying the sparse pre-conditions (e.g. only sorted lists). Property-based generators are hard to write and to keep up to date as definitions change. Given the effort required to set up the proper testing infrastructure, a potential QuickChick user will rightfully wonder whether testing is really worth the trouble for her particular problem, or whether she can make faster progress by immediately attempting a proof and finding the bugs that way (which can of course be very costly). So in order for testing to reduce the cost of formal verification, the cost of the testing itself has to be relatively low. On the other hand, our experience shows that dependable testing can rarely be fully automated and the user still needs effective ways of interacting with the testing framework.

Objective 1: Reduce PBT cost while maintaining full generality and programmability The first and most important objective of this project is to reduce the human effort required for using PBT during the normal Coq proving process. The main technical innovation we propose is Luck, a novel domain-specific language for property-based generators that borrows ideas from functional logic programming and constraint solving. Syntactically, we use a simple lambda calculus with algebraic datatypes extended with “unknown values” (logical variables). When all values a Luck program uses are known, the program evaluates in the standard way, producing a single result. When some values used by the program are unknown, Luck randomly instantiates these unknowns to the extent needed [1]. Wrong instantiations can cause costly backtracking, so we allow the user to delimit expressions for which instantiation should be delayed and for which our language gathers and propagates constraints before committing to an instantiation. For instance, if `x` is an unknown in the expression `low < x && x < high` we only want to instantiate `x` after processing both constraints. To account for instantiation and backtracking the semantics of Luck expressions is defined in terms of probabilistic coinductive computation trees. Work is already quite advanced on defining this semantics, on working out the metatheory, and on building a prototype Luck interpreter [13, 9]. Still, more research is needed before Luck is ready for mainstream usage. We plan to build an efficient compiler for Luck, to automatically translate Coq inductive relations to Luck expressions, and to devise an optimal parameter selection framework for Luck generators, which will allow users to express their testing objectives in higher-level terms and use algorithmic optimization to choose good weights for local probabilistic choices.

Challenge 2: Property-based testing could use stronger formal foundations While usually very effective in practice, PBT does not have any explicit formal foundation. How do we know that user provided executable code is testing the right logical proposition? Does a property-based generator for property `P` only produce data satisfying `P` and even more importantly can it in principle produce *all* data satisfying `P`? Integrating property-based testing with Coq creates an interesting opportunity: it allows us to provide formal answers to such questions by verifying the testing code using Coq itself. In recent work [11] we have introduced a foundational verification framework for QuickChick, in which the semantics of a generator is the set of values that have non-zero probability of being generated. Building on this, we assign a semantics to each checker expressing the logical proposition it tests, abstracting away from the computational cost and the precise probability distributions of the generators it uses. Our framework is firmly grounded in a fully verified implementation of QuickChick itself, using the same underlying verification methodology. We have also applied our methodology to a complex case study on testing an information-flow control abstract machine [8], demonstrating that our verification methodology is modular and scalable and that it requires

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

minimal changes to existing code.

Objective 2: Formally verify testing code using the proof assistant Testing errors can conceal important bugs and thus reduce its benefits, and are especially hard to find and debug in the presence of randomness—generators are probabilistic programs. While the framework and the possibilistic abstraction above are a good start at preventing testing errors and at providing stronger formal foundations to testing, there are many technical challenges remaining to be solved. First, verification in our framework is at the moment a manual process, still we believe the sets of outcomes abstraction is highly suitable for automation. We will extend our framework to automatic verification using SMT solvers, making it much more practical. Second, while our sets of outcomes semantics greatly simplifies reasoning so that we can scale up to interesting examples, it is still a rather coarse abstraction, that will miss inherently probabilistic bugs. We will extend our framework to probabilistic reasoning and probabilistically verify our QuickChick implementation. More ambitiously, we will verify the correctness of the Luck interpreter, first with respect to the sets of outcomes abstraction, and then in the probabilistic setting.

Challenge 3: Evaluating testing quality and obtaining confidence from this is difficult As Dijkstra famously stated: "Testing shows the presence, not the absence of bugs". In particular, using PBT and not finding any bugs does not mean there are none left. The property under test could of course be correct, but one could also be testing it poorly, and improving the testing could lead to more bugs being found. Still, how do we know that something is wrong with our testing and in what way to improve it? And how do we know when to finally stop improving testing and start proving because we have high confidence in testing? Formal verification using the framework from Objective 2 provides a sanity check that can eliminate a whole class of testing bugs, but it only guarantees that the checkers and generators are testing the right property; it does not ensure that they are testing it *well*, so passing this sanity check is useful but cannot on its own provide strong confidence in the thoroughness of testing. Similarly, gathering statistics and checking code coverage are useful techniques for estimating the quality of testing, but it is hard to obtain significant confidence just by looking at these. We want to propose a more systematic, and potentially more reliable technique for evaluating the quality of testing, something that is not only an indicator of bad testing, but also an indicator of good testing. Testing is good when it finds bugs, so we systematically introduce bugs and make sure they are reliably found.

Objective 3: Evaluate thoroughness of testing by systematically introducing bugs The integration with Coq raises a new opportunity for evaluating the thoroughness of PBT. The main idea of what we call *polarized mutation testing* is to use the declarative Coq definitions to systematically mutate the artifact under test to introduce all pointwise bugs from an interesting class and to make sure that they are all found by testing. Once all introduced bugs are found and no new bugs are discovered in the non-mutated artifact we can obtain higher confidence that indeed no bugs are left. The main novelty over previous mutation testing work is that instead of blindly introducing syntactic changes that do not necessarily violate the tested property and waste precious human effort weeding them out, we only introduce real bugs by exploiting the logical structure of the Coq property and the declarative description of the artifact. If the tested property is tight (e.g., having no unnecessary precondition) then strengthening the predicates appearing in positive positions or weakening the predicates in negative ones is guaranteed to only introduce real bugs. For instance, we can add bugs to type progress by strengthening the step relation (e.g. dropping whole stepping rules) and to noninterference by weakening it (e.g. dropping information-flow side-conditions). Similarly, we can break type preservation either by strengthening the occurrence of the typing relation in the conclusion, or by weakening the occurrence in the premise.

Preliminary mutation experiments with increasingly complex information-flow monitors [8], the simply-typed λ-calculus, and CompCert have been very encouraging. We discovered for instance that a generator for simply-typed λ-calculus terms [10] was not generating shadowed variables, which caused it to miss a capturing bug in substitution. We also discovered that CSmith, a highly successful tool for testing C compilers [14], could not detect a bug we introduced in CompCert, causing it to perform tail-call optimization even for non-tail-recursive functions. We will work out these experiments in detail and perform new ones. In particular, we will apply this idea to test the soundness of the F* type-checker [12], by systematically strengthening the specifications of well-typed programs. Longer term, we will develop a general methodology and theory of polarized mutation testing, and integrate it with the rest of QuickChick. We will also use this methodology to assess the success of Luck generators and parameter optimization framework (Objective 1).

Challenge 4: Our solutions need to work well in practical scenarios The problems we attack in this proposal are practically motivated, so the solutions we are looking for need to be practical.

Objective 4: Perform realistic case studies We will measure success by performing several realistic case studies targeting widely-used TLS implementations [3], secure compilers, tag-based reference monitors [2, 8], and complex type-checkers, including F⋆ [12] and Coq. The TLS case study is representative of a large-scale industrial problem, while the others are representative to the kind of problems current Coq users face daily.

References

[1] S. Antoy. A needed narrowing strategy. *JACM*. 2000.

[2] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. *Oakland S&P*. 2015.

[3] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan. FLEXTLS: A tool for testing TLS implemen-

tations. In *9th Workshop on Offensive Technologies, WOOT '15*. 2015. **Best Paper Award**.

[4] L. Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. *CPP*. 2012.

[5] H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. *ACL2*, 2011.

[6] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ICFP*. 2000.

[7] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. *TPHOLs*. 2003.

[8] C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky, A. Azevedo de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis. Testing noninterference, quickly. *ICFP*. 2013. Extended version submitted to special issue of JFP, September 2014.

[9] L. Lampropoulos, B. C. Pierce, C. Hriţcu, J. Hughes, Z. Paraskevopoulou, and L. Xia. Making our own Luck: A language for random generators. Draft, 2015.

[10] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*. 2011.

[11] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. *ITP*. 2015.

[12] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, and J.-K. Zinzindohoue. Dependent types and multi-monadic effects in F*. 2015. To appear in POPL 2016.

[13] L. Xia and C. Hriţcu. Integrating functional logic programming with constraint solving for random generation of structured data. Inria Internship Report, 2015.

[14] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011.