The COGENT Case for Property-Based Testing

Zilin Chen Liam O'Connor Gabriele Keller Gerwin Klein Gernot Heiser
Data61 and UNSW, Australia
first.last@data61.csiro.au

Abstract

Property-based testing can play an important role in reducing the cost of formal verification: It has been demonstrated to be effective at detecting bugs and finding inconsistencies in specifications, and thus can eliminate effort wasted on fruitless proof attempts. We argue that in addition, property-based testing enables an incremental approach to a fully verified system, by allowing replacement of automatically generated tests of properties stated in the specification by formal proofs. We demonstrate this approach on the verification of systems code, discuss the implications on systems design, and outline the integration of property-based testing into the COGENT framework.

CCS Concepts • **Software and its engineering** **Software testing and debugging**; *Formal software verification*; Functional languages;

Keywords COGENT, QuickCheck, Refinement, Systems software, Formal methods

ACM Reference Format:
Zilin Chen Liam O'Connor Gabriele Keller Gerwin Klein Gernot Heiser. 2017. The COGENT Case for Property-Based Testing. In *Proceedings of PLOS'17: 9th Workshop on Programming Languages and Operating Systems (PLOS'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3144555.3144556

1 Introduction

COGENT [2, 13, 30] is a restricted, purely functional language aimed at reducing the cost of formally verifying systems code. COGENT grew out of the experience of the verification of the seL4 microkernel [23]. Specifically, we had observed that many of the more low-level proofs in the chain connecting the high-level specification with the C implementation are time consuming and tedious, but not particularly involved, and seemed good candidates for automation.

PLOS'17, October 28, 2017, Shanghai, China
© 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5153-9/17/10.
https://doi.org/10.1145/3144555.3144556

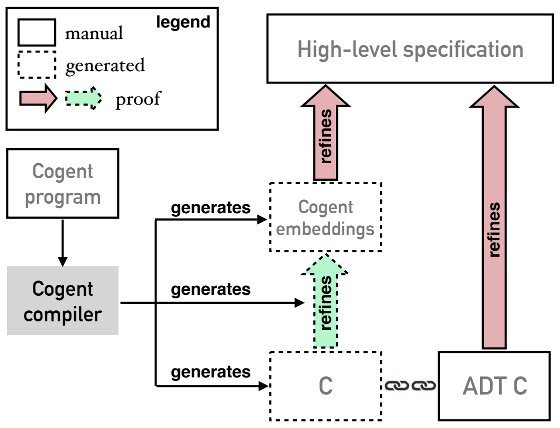

Figure 1. An overview of COGENT

COGENT makes such automation possible by abstracting over low-level issues, such as explicit memory management. As Figure 1 shows, the COGENT compiler generates C code, several embeddings of the COGENT semantics in the interactive theorem prover Isabelle [28], and proofs connecting the two. Similar to Rust [32], COGENT is equipped with a linear type system to allow destructive updates while retaining the purely functional semantics. Because of the linear type system, it is impossible to implement data structures in COGENT which rely even temporarily on sharing. These have to be implemented in C, verified separately, and imported as abstract data types (ADTs) with an interface that observes the linear type constraints.

While the compiler-generated automatic proof reduces the gap between the high-level specification and the C implementation, the remaining gap — manually connecting the COGENT semantics to the high-level specification, as well as the verification of the manually-written C components — still requires significant time and effort from developers.

Property-based testing is a promising technique for reducing this cost. Similar to formal verification, property-based testing uses a specification of the desired properties of a unit under test. From this specification it automatically generates test cases to search for counter-examples. Hughes [21] and Arts et al. [4] show that property-based testing is effective for detecting bugs and finding inconsistencies in specifications. In their work on secure information flow, Hriţcu et al. [20] observe that property-based testing is especially valuable in formal verification, as it can eliminate the wasted effort of trying to prove a faulty or ill-specified system correct.

In this paper, we posit that there are further reasons why property-based testing is well suited to work in tandem with formal verification: we argue that the costs and downsides of using property-based testing apply to a much lesser degree in the context of verification, and discuss how property-based testing can be used to facilitate the verification process. We present an approach to using property-based testing in the context of systems code verification. Using a worked example, we outline our work in progress implementing property-based testing support for COGENT, and show how this fits into the systems verification framework. Furthermore, we outline the challenges we identified and our plans for future work.

2 Rationale

In a property-based testing framework, testers specify the logical pre- and postconditions of functions, and the framework generates random test data and runs each individual test over and over with different inputs, searching for a counter-example. Property-based testing is effective but not universally applicable — in practice it is usually hard to describe the complete system in terms of logical properties. In the context of formal verification, however, we will have to model the complete behaviour of the system in logic in any case. By using the same set of properties for both testing and verification purposes, it is possible to first check properties by extensive, automated testing, and only attempt to prove them once we are reasonably confident they actually hold.

We can also check whether the set of properties we establish is sufficient to connect the implementation to the high-level specification, by using these properties as so-far unproved (but validated by tests) candidate theorems. Hence, combining the techniques enables a more incremental approach than the all-or-nothing we get with formal verification alone. While non-exhaustive testing can never guarantee the absence of errors, we can use property-based testing to establish confidence in the correctness of a program, and then increase the level of confidence with each verified component.

It is useful to keep the same set of properties under property-based testing even once formal verification is complete. When the program inevitably evolves, the testing approach will more quickly tell us which properties break and (optimistically) which are likely to still hold. While the formal verification can also be replayed, and will tell us which properties definitely still hold, it will fail conservatively when the proof breaks or changes, i.e. when the reason for a property to hold has changed, not only when the property itself is no longer true.

Purely functional languages, like COGENT, are particularly suitable for property-based testing as well as verification, since the behaviour of a function only depends on the input values, and not on any hidden state. Unfortunately, just the fact that a system is written in a purely functional language does not necessarily mean it can be validated using property-based testing. It is still vital to design the system carefully. For example, if a function only accesses a small portion of the state, only those portions should be passed into the function. Otherwise, it is unlikely that running tests on randomly generated test data achieves sufficient code coverage, as many of the variations made to the input may have no effect on the behaviour of the function.

In practice, this means that to obtain a system that is suitable for property-based testing, special considerations are needed in the design; in general, a naive translation of C code into COGENT will be too dependent on (mostly incidental) global state. While this seems to be a high price to pay, a verification-friendly design also requires us to carefully modularise and compartmentalise the state accessible from each part of the system [1, 3], so in fact, these design restrictions impose no additional burden in the context of COGENT.

Claessen and Hughes [11] introduced property-based testing with QuickCheck, a combinator library for the functional language Haskell. By now, frameworks which implement at least some part of the QuickCheck functionality are available for most general-purpose programming languages. Instead of implementing a new framework for COGENT, we are providing an interface to use the Haskell QuickCheck library, as this is currently the most mature and fully-featured implementation of property-based testing, and it integrates well with the COGENT compiler.

In the remainder of the paper, we first introduce our approach to both testing and verifying file systems using refinement (Section 3.1), illustrate how our QuickCheck architecture enables the incremental development of fully verified systems code (Section 3.2), and show an example of applying our framework to a COGENT ADT library (Section 3.3). After that, we discuss a number of major issues that we encountered during development and our proposed solutions (Section 4). Finally, we briefly summarise related work (Section 5) and conclude.

3 QuickChecking COGENT

Existing verification work in COGENT, such as the verification of the file system BilbyFs [2], expresses the functional correctness of the system as a *refinement* statement from an abstract correctness specification. Our framework, therefore, allows such refinement statements to be expressed as QuickCheck properties. We start with a brief introduction to the notion of refinement.

3.1 Using refinement for testing and verification

As mentioned in Section 2, COGENT's purely functional semantics provides a simple formal model of a program's behaviour. Specifically, it enables reasoning about programs using *equational* principles. Fortunately, the property languages used in most property-based testing libraries have a similarly equational semantics. The remaining challenge then is to express our notion of what constitutes a *correct* program with formulae that can be either proved using equational reasoning or validated by property-based testing.

As stated above, functional correctness as used for COGENT programs uses a *refinement statement* from an *abstract specification*. It is *abstract* in the sense that all details not relevant to program correctness are omitted (for example, no attention is paid to performance or computability in the abstract specification).

In an imperative setting, a simple model for both the abstract specification and the concrete implementation would be relations on states, describing every possible behaviour of the program as the manipulation of some global state. A program C is a *refinement* of a program A if every possible behaviour in the model of C is observable in that of A, that is $[\![C]\!] \subseteq [\![A]\!]$. This means that if we prove a property about every execution for our abstract specification, we know that the property holds for all executions of our concrete implementation.

An abstract specification typically has a different, smaller state space than its concrete implementation, so the simple subset relationship does not quite capture what we require of refinement; we need a notion of correspondence of states. We get this from some additional machinery from the world of *data refinement*. We introduce a *refinement relation* R that relates abstract and concrete states, and show that each step our program takes preserves this relation; the relation represents the desired correspondence. We must show that our abstract program behaves analogously to the concrete program given corresponding initial states. That is, if R relates our initial abstract and concrete states, then every final state of a concrete operation conc will be related by R to a final state of a corresponding abstract operation abs:

$$R; [\![\mathsf{conc}]\!] \subseteq [\![\mathsf{abs}]\!]; R \qquad \begin{tikzcd} \circ \arrow[r, Rightarrow, "\mathsf{abs}"] \arrow[d, dash, dashed, "R"'] \arrow[dr, phantom, "\subseteq"] & \circ \arrow[d, dash, dashed, "R"] \\ \bullet \arrow[r, Rightarrow, "\mathsf{conc}"'] & \bullet \end{tikzcd}$$

where ";" is forward composition of relations. A theorem like this for each operation in our program forms the bulk of a *forward simulation* proof of data refinement [15].

In COGENT, however, there is no global state. COGENT is a purely functional, deterministic, total language, and models all functions as plain mathematical functions. In such a scenario, the only state involved consists of the inputs and outputs to the function, simplifying the refinement statement. Given an abstract function $\mathsf{abs} :: X_a \to Y_a$, and a concrete COGENT function $\mathsf{conc} :: X_c \to Y_c$, then, assuming the existence of refinement relations R_X and R_Y, we can express the statement that conc refines abs as:

$$R_X\ i_a\ i_c \implies R_Y\ (\mathsf{abs}\ i_a)\ (\mathsf{conc}\ i_c)$$

This, however, places unnecessary constraints on our abstract specification. While COGENT is deterministic and total, our

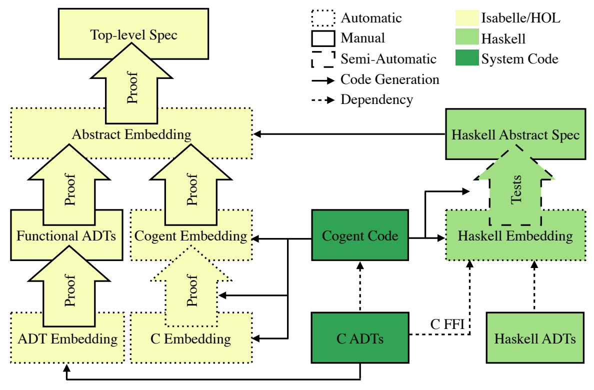

Figure 2. Overview of the QuickCheck framework

abstract specification need not be. In fact, it is often desirable to allow non-determinism to reduce the complexity of the abstract specification. We model non-determinism by allowing abstract functions to return a *set* of possible results. Then, our refinement statement merely requires that the single concrete result correspond to one of the possible abstract results:

$$R_X\ i_a\ i_c \implies \exists o_a \in \mathsf{abs}\ i_a.\ R_Y\ o_a\ (\mathsf{conc}\ i_c)$$

Defining the notation

$$\mathit{corres}\ R\ a\ c \overset{\text{def}}{=} \exists o \in a.\ R\ o\ c$$

allows us to clean up this form of refinement statement:

$$R_X\ i_a\ i_c \implies \mathit{corres}\ R_Y\ (\mathsf{abs}\ i_a)\ (\mathsf{conc}\ i_c)$$

The theorems that capture correctness for COGENT systems typically have this *corres* format. Our goal, then, is to encode these properties in Haskell as machine-testable properties.

3.2 The QuickCheck architecture for COGENT

Our COGENT QuickCheck framework (Figure 2) extends the existing COGENT compiler with additional code generation features, allowing a Haskell embedding of the COGENT code to be generated, as well as the refinement properties themselves. These refinement properties require the user to supply Haskell definitions for all refinement relations, as well as an abstract specification.

As previously mentioned, COGENT's design requires that some components (mostly ADT implementations that are common to multiple file system implementations) are not implemented in COGENT, but in C, and accessed via COGENT's foreign function interface (FFI). In the Haskell embedding, the programmer either interfaces the same implementations via Haskell's C FFI, or implements these ADTs in Haskell. Currently, this is done manually, but the compiler could generate the bindings to the C calls; this is left for future work.

These ADTs, which are not implemented in COGENT, have to be verified separately at some point to obtain a fully verified system. In the mean time, QuickChecking these components can provide us with a higher degree of confidence of their correctness, as we shall see in Section 3.3.

Once the tester has supplied our framework with the necessary definitions, they can use QuickCheck to test whether the generated Haskell embedding is a refinement of the Haskell specification. To facilitate the testing process, and possibly reflect the structure of the proofs, the tester can also provide multiple, increasingly abstract specifications. Our framework can generate Isabelle versions of these Haskell specifications which can then be used for manual proof. This ensures that the two specifications can remain in sync, allowing the user to move between proving and testing as needed. Lastly, the proof engineer must prove that the most abstract Haskell specification is a refinement of the top-level correctness specification in Isabelle. Typically, this proof is straightforward as there is very little difference between the two.

The top-level Isabelle specification is written in higher-order logic, and is not executable. It is highly non-deterministic, in order to model behaviours of the system which are dependent on inputs out of our control. For example, the system memory allocator can non-deterministically fail when it runs out of memory, and disk device drivers can non-deterministically fail in a variety of ways, such as when the disk is full or an I/O failure occurs.

In the Haskell specification, we must also model this non-determinism, however the Haskell specification is more constrained than that of Isabelle: it must be *executable*. In general, simulating a non-deterministic model can be exponential in time and space, compared to a deterministic version.

We address this by modelling only a minimal amount of non-determinism in our Haskell specification, using a similar non-deterministic set monad as in our Isabelle specification. By keeping the amount of non-determinism low, we reduce the cost of modelling all possible results to manageable levels.

Finally, in addition to the specifications and properties, the generation of test data is also an important ingredient in a property-based testing framework. While the COGENT compiler will automatically provide *test data generators* for all COGENT types used in the program, the user must provide their own generators for any abstract data types used in the Haskell embedding or specification.

3.3 Example: testing refinement of an ADT library

To test a property with QuickCheck, we can encode it as a Boolean function.[1] When we pass the property to the `quickCheck` function provided by the QuickCheck library, it will apply the Boolean function to a number of randomly generated input data. If it evaluates to `False` for any test case, QuickCheck has found a counter-example. It will then try to shrink the test case to find the simplest counter-example and report that to the user. If it cannot find a counter-example, it will report that the tests have run successfully.

[1]It does not have to be a Boolean function — QuickCheck is more flexible.

In our case, we want to test for each function whether it observes the correspondence theorem from Section 3.1. Let us take an ADT function `ba_create` — which allocates and initialises a byte array `BA` of a given length — as an example and showcase the development process. [2]

```
type St
type BA
type R a b = <Success a | Error b>

ba_create : (St, U32) -> R (St, BA) St
```

In the above COGENT code, type `St`, `BA` and function `ba_create` are all abstract; their definitions are written in C. `St` models the global system state, which is threaded through the `ba_create` function because it allocates memory and thus changes the global state. `ba_create` returns a tagged union type `R (St, BA) St`, which is either a new system state paired with the newly created byte array, or in the error case, a new system state alone.

A Haskell embedding of the code above is generated by the COGENT compiler. This embedding serves as the non-deterministic specification, which is refined by the C implementation:

```
type St = ???
type BA = ???
data R a b = Success a | Error b

hs_ba_create :: (St, U32) -> ND (R (St, BA) St)
hs_ba_create = ???
```

The tester needs to fill in the definitions for the abstract datatypes and functions, denoted by "???" in the Haskell template:

```
type St = ()
type BA = [Byte]

hs_ba_create :: (St, U32) -> ND (R (St, BA) St)
hs_ba_create ((),len) =
  (return $ Success ((), replicate len 0))
    `alternative`  (return $ Error ())
```

The type constructor `ND` is the non-determinism monad which contains all the possible results of a computation. The monadic `return` operator lifts a result to a singleton set; `alternative` simply takes the union of its both operands. In this case, the result is a two-element set {`Success ((), replicate len 0)`, `Error ()`}. This non-determinism is present because the state of the external world has been abstracted to a single unit type in our Haskell embedding.

In order to call the COGENT compiled C code from Haskell, we make use of Haskell's FFI facilities:

```
foreign import ccall unsafe "ba_create"
  c_ba_create' :: Ptr Ct4 -> IO (Ptr Ct5)

c_ba_create :: Ct4 -> IO Ct5
```

[2]This example is adapted from our word array library for presentation.

```
c_ba_create a = peek =<< c_ba_create' =<< new a
```

The `foreign import ccall` declaration expresses that the Haskell function `c_ba_create'` is the interface to C function `ba_create`. `Ct4` and `Ct5` are the Haskell representations of the compiler-generated C types; their definitions can be generated using FFI tools like hsc2hs [18] or c2hs [9]. `c_ba_create` is merely a wrapper to hide the pointers.

To compare the behaviour of `c_ba_create` and `hs_ba_create`, we need a Haskell definition of the refinement relations for both the inputs and the outputs of these two functions. In future, we hope to generate these automatically with the aid of a separate domain-specific language. Depending on the property under test, it can be convenient to represent one or more of these refinement relations as an *abstraction function*, which rather than merely relating two data types, actually converts the concrete data type into its abstract equivalent. In our example, we define an abstraction function for the inputs, and a refinement relation for the outputs (the definitions are omitted for presentation):

```
abs_ba_create_arg :: Ct4 -> (St, U32)
rel_ba_create_ret :: R (St, BA) St -> Ct5
                  -> IO Bool
```

In general, if a Haskell representation of a C type consists of pointers, it must be wrapped in an `IO` monad when we want to manipulate these pointers. That is why function `rel_create_ret` returns an `IO Bool`, as `Ct5` happens to include pointers, whereas `Ct4` does not.

Now, as a final step, we have to relate the two implementations, `hs_ba_create` and `c_ba_create` via a function which models the correspondence property from Section 3.1.

```
prop_ba_create_corres = monadicIO $
  forAllM gen_Ct4 $ \ic -> run $ do
    oc <- c_ba_create =<< ic
    oa <- return $ hs_ba_create
            (abs_ba_create_arg ic)
    corresM rel_ba_create_ret oa oc
```

Slightly simplified, this function can be read as: for any type correct concrete input `ic` (generated by test data generator `gen_Ct4`) and its abstraction, check that the results of applying the concrete function `c_ba_create` and the abstract function `hs_ba_create` respectively are related via the refinement relation `rel_ba_create_ret`. The `corresM` function is a slight generalisation of our *corres* notation to allow the refinement relation to be defined inside the `IO` monad, as the C function is called via the FFI and is thus impure:

```
corresM :: (a -> c -> IO Bool) -> ND a -> c
        -> IO Bool
corresM rrel ma c = anyM (`rrel` c) $ toList ma
```

Running the tests by passing `prop_ba_create_corres` to the `quickCheck` function, we get

```
*ByteArray> quickCheck prop_ba_create_corres
+++ OK, passed 100 tests.
```

4 Discussion

At the time of writing, we have applied our prototype framework to series of small examples, some of which are derived from the BilbyFs project [1], such as a subset of the word array ADT library and some simple file system operations.

Testing ADT code is different from testing plain COGENT code, as they have different abstraction levels and different refinement steps. For ADTs, as COGENT does not automatically prove any refinement steps, the refinement under test needs to extend all the way from the C semantics to the abstract specification, while for non-ADTs, the Haskell embedding of COGENT is the lowest level of abstraction that is necessary to test. The examples we drew from a real file system, while simple, involve a wide range of datatypes, some of which are from the Linux kernel. Testing these examples has given us an idea of how much boilerplate code is required and which components of the testing infrastructure can be automatically generated. Since real file systems code inevitably accesses a wide range of ADTs and wrapper code from the kernel, it also tests the ability to integrate these different units together.

4.1 Modular testing

Our QuickCheck machinery does not require the user to test the entire system as a whole. Instead, the user may test refinement for each function or for a cluster of functions at a time. Typically, the ADTs implemented in C are a common module, shared across many systems. Accordingly, our framework allows for developers to test their ADTs in isolation, without regard to their greater context within the overall system. This modularity is aided by COGENT's purely functional semantics.

Once the ADT libraries have been implemented in C, tested with our framework, and even verified with Isabelle, they can then be linked with the rest of the COGENT system, including with our testing framework. Alternatively, they may choose to replace these C implementations with native Haskell ones for the purposes of testing the main COGENT code. Our framework supports both approaches.

4.2 Enforcing COGENT's contract

ADTs that are used from within COGENT are free to make use of destructive update, however the linear type system that prevents these mutations from being distinguished from pure functions is no longer present in the Haskell embedding. This means that extra care must be taken to use these functions in a linear way, even in the Haskell abstract specifications. A possible future version of our framework could use GHC Haskell's new linear types extension [5] to prevent mistakes of this kind from being made.

4.3 Improving the test data generation

QuickCheck comes with a set of predefined random test data generators for a wide range of built-in Haskell types. It even comes with the ability to derive new data generators for user-defined datatypes [26]. One feature that would be desirable is to guide test data generation using domain knowledge. Ideally, generators should produce test data that satisfies the preconditions of the tested unit. This means that time is not wasted testing spurious or invalid inputs. For example, when we test array initialisation functions, the size of allocated memory is represented as a 32-bit integer, but testing the full range of inputs would be impractically slow, as massive data structures would be repeatedly allocated, copied, and marshalled between the Haskell and C heaps.

COGENT also supports higher-order functions, and our test framework supports generating functions, however in a very different way than the basic QuickCheck library. QuickCheck will generate essentially an arbitrary mapping from the function's domain to its codomain, whereas our framework is able to exploit a quirk in the way that COGENT compiles first-class functions. Rather than represent a function value as a function pointer, we keep track of all possible functions of a given type, and assign them a particular index. The function value is therefore just represented as single machine word containing that index. This representation is chosen because it interoperates more cleanly with our existing C verification infrastructure. Happily, this means we can simply generate a random index and choose from all of the available functions of the required type. This makes our testing more intelligent, as the only functions provided as arguments to higher-order functions are those that actually exist in the program, however it does mean that such higher-order functions cannot be tested in complete isolation from the rest of the system.

4.4 Testing kernel modules

A file system is typically compiled as a kernel module and run in kernel mode, but our test framework runs in user mode. This discrepancy means that, for our prototype, we have ported our file systems code to run in user mode. This is a fairly common practice anyway in systems programming, with libraries such as FUSE [16] facilitating the user-space execution of kernel modules. A possible alternative is to use a system such as House [19] or HaLVM [22] to run Haskell in kernel mode, allowing us to make our testing environment resemble the real run-time environment of the software more closely.

4.5 Future work

The COGENT QuickCheck framework is very much work in progress: the tester still has to write a significant amount of code which could be generated by the compiler. This includes Haskell datatypes and functions for the FFI to C, as well as boilerplate code for *corres* instances, refinement relations and abstraction functions. Generating refinement relations and abstraction functions requires domain knowledge, and therefore would require some sort of domain-specific language to instruct the generator on how to abstract the types.

Chen [10] proposes an alternative to integrating Haskell into our COGENT framework: a high-level language extension to COGENT could meet our needs without having to build many of the aforementioned auxiliaries. With such a language, COGENT users would have a more pleasant experience, and they would no longer need to learn any new language other than the extended COGENT.

We also intend to improve the data generator and shrinking algorithm in the QuickCheck framework, which will pave the way for us to do a more complete case study on the file systems that we developed. We see some links between the properties specified in property-based testing and in Isabelle. Whether one can be generated from the other, and whether the testing (more precisely, counter-examples) can guide the theorem prover to automate the manual proofs, are open research questions.

5 Related Work

External property-based testing libraries have been connected to many proof assistants, including Isabelle/HOL, as a way to generate quick counter-examples for proposed lemmas [7, 17]. In principle, this would provide an alternative implementation to our design: one could generate the Isabelle/HOL semantics of the COGENT program and run Isabelle Quickcheck on the formalisation instead of the COGENT program itself. The main difference is performance and implementation effort. Our approach takes advantage of the more mature Haskell environment for QuickCheck as well as the existing compiler infrastructure for COGENT that is written in Haskell to achieve better performance, which translates into a higher number of test cases, potentially with hardware in the loop. For using Isabelle QuickCheck directly, one would have to replicate the test harness and test case generators in ML for no immediate benefit in this context.

Some property-based testing frameworks are formally verified and implemented in proof assistants [29, 31]. These frameworks shed light on how to verify that a property-based testing library actually tests the desired properties.

QuickCheck has been used for testing of a variety of high-level properties, such as information flow control [14, 20], mutual exclusion [12], and functional correctness of AUTOSAR components [4, 27]. As far as we know, ours is the first framework to use property-based testing for refinement-based functional correctness statements.

There is a wide range of work on generating test data more intelligently [6, 8, 24], as well as approaches to test data shrinking that preserve invariants about the generated data [25, 33]. These approaches are highly complementary to our framework. Our refinement framework should provide a good testing ground to evaluate these techniques.

References

[1] Sidney Amani. 2016. *A Methodology for Trustworthy File Systems*. PhD Thesis. CSE, UNSW, Sydney, Australia.

[2] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *ASPLOS*. Atlanta, GA, USA, 175–188.

[3] Sidney Amani and Toby Murray. 2015. Specifying a Realistic File System. In *Workshop on Models for Formal Analysis of Real Systems*. Suva, Fiji, 1–9.

[4] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. 1–4. https://doi.org/10.1109/ICSTW.2015.7107466

[5] Jean-Philippe Bernardy, Mathieu Bosepflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Retrofitting Linear Types. (2017). https://www.microsoft.com/en-us/research/wp-content/uploads/2017/03/haskell-linear-submitted.pdf.

[6] Achim D. Brucker and Burkhart Wolff. 2013. On theorem prover-based testing. *Formal Aspects of Computing* 25, 5 (2013), 683–721. http://dx.doi.org/10.1007/s00165-012-0222-y

[7] Lukas Bulwahn. 2012. The New Quickcheck for Isabelle: Random, Exhaustive and Symbolic Testing Under One Roof. In *Proceedings of the Second International Conference on Certified Programs and Proofs (CPP'12)*. Berlin, Heidelberg, 92–108. http://dx.doi.org/10.1007/978-3-642-35308-6_10

[8] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. 2012. A First Step in the Design of a Formally Verified Constraint-Based Testing Tool: FocalTest. *Tests and Proofs: 6th International Conference, TAP 2012, Prague, Czech Republic, May 31 – June 1, 2012. Proceedings* (2012), 35–50.

[9] Manuel M. T. Chakravarty. 1999. C -> HASKELL, or Yet Another Interfacing Tool. In *Implementation of Functional Languages, 11th International Workshop, IFL'99, Lochem, The Netherlands, September 7-10, 1999, Selected Papers*. 131–148. https://doi.org/10.1007/10722298_8

[10] Zilin Chen. 2017. COGENT$^{\Uparrow}$: Giving Systems Engineers A Stepping Stone (Extended abstract). In *The workshop on Type-Driven Development (TyDe'17)*. https://www.cse.unsw.edu.au/~zilinc/tyde17.pdf.

[11] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *5th ICFP*. New York, NY, USA, 268–279. http://doi.acm.org/10.1145/351240.351266

[12] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. 2009. Finding Race Conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. New York, NY, USA, 149–160. http://doi.acm.org/10.1145/1596550.1596574

[13] Cogent 2017. COGENT Homepage. (2017). http://ts.data61.csiro.au/projects/TS/cogent.pml.

[14] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2014. A Verified Information-Flow Architecture. In *POPL*. San Diego, CA, USA, 165–178.

[15] Willem-Paul de Roever and Kai Engelhardt. 1998. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts Theoretical Comp. Sci. United Kingdom.

[16] FUSE Developers. 2017. The FUSE Project. https://sourceforge.net/projects/fuse. (2017).

[17] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. 2003. Combining Testing and Proving in Dependent Type Theory. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLS '03)*. Berlin, Heidelberg, 188–203. http://dx.doi.org/10.1007/10930755_12

[18] GHC Team. 2017. *GHC User's Guide Documentation, Version 8.2.1*. User's Guide. 463–466 pages. https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf.

[19] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. 2005. A principled approach to operating system construction in Haskell. In *10th ICFP*. Tallinn, Estonia, 116–128.

[20] Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. New York, NY, USA, 455–468. http://doi.acm.org/10.1145/2500365.2500574

[21] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 169–186. https://doi.org/10.1007/978-3-319-30936-1_9

[22] Galois Inc. 2017. The Haskell Lightweight Virtual Machine (HaLVM) source archive. https://github.com/GaloisInc/HaLVM. (2017).

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *SOSP*. Big Sky, MT, USA, 207–220.

[24] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. New York, NY, USA, 114–129. http://doi.acm.org/10.1145/3009837.3009868

[25] David R. MacIver. 2016. *Integrated vs Type-based Shrinking*. Article. http://hypothesis.works/articles/integrated-shrinking.

[26] Neil Mitchell. 2017. The derive package. (2017). https://hackage.haskell.org/package/derive.

[27] Wojciech Mostowski, Thomas Arts, and John Hughes. 2017. Modelling of Autosar Libraries for Large Scale Testing. In *Proceedings 2nd Workshop on Models for Formal Analysis of Real Systems, MARS@ETAPS 2017, Uppsala, Sweden, 29th April 2017*. 184–199. https://doi.org/10.4204/EPTCS.244.7

[28] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics with Isabelle/HOL*.

[29] Liam O'Connor. 2016. Applications of Applicative Proof Search. In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*. New York, NY, USA, 43–55. http://doi.acm.org/10.1145/2976022.2976030

[30] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement Through Restraint: Bringing Down the Cost of Verification. In *ICFP*. Nara, Japan.

[31] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *ITP 2015 - 6th conference on Interactive Theorem Proving (Lecture Notes in Computer Science)*, Vol. 9236. Nanjing, China. https://hal.inria.fr/hal-01162898

[32] Rust 2014. The Rust Programming Language. http://rustlang.org. (2014). Accessed March 2015.

[33] Jacob Stanley. 2017. *Hedgehog will eat all your bugs*. Open Source Project. https://github.com/hedgehogqa/haskell-hedgehog.