Property-Based Testing: Climbing the Stairway to Verification

Zilin Chen
UNSW Sydney
Sydney, Australia
zilin.chen@student.unsw.edu.au

Christine Rizkallah
University of Melbourne
Melbourne, Australia
christine.rizkallah@unimelb.edu.au

Liam O'Connor
University of Edinburgh
Edinburgh, UK
l.oconnor@ed.ac.uk

Partha Susarla
Melbourne, Australia
partha@spartha.org

Gerwin Klein
Proofcraft and UNSW Sydney
Sydney, Australia
kleing@unsw.edu.au

Gernot Heiser
UNSW Sydney
Sydney, Australia
gernot@unsw.edu.au

Gabriele Keller
Utrecht University
Utrecht, Netherlands
g.k.keller@uu.nl

Abstract

Property-based testing (PBT) is a powerful tool that is widely available in modern programming languages. It has been used to reduce formal software verification effort. We demonstrate how PBT can be used in conjunction with formal verification to incrementally gain greater assurance in code correctness by integrating PBT into the verification framework of Cogent—a programming language equipped with a certifying compiler for developing high-assurance systems components. Specifically, for PBT and formal verification to work in tandem, we structure the tests to mirror the refinement proof that we used in Cogent's verification framework: The expected behaviour of the system under test is captured by a functional correctness specification, which mimics the formal specification of the system, and we test the refinement relation between the implementation and the specification. We exhibit the additional benefits that this mutualism brings to developers and demonstrate the techniques we used in this style of PBT, by studying two concrete examples.

CCS Concepts: • **Software and its engineering** **Software testing and debugging**; **Functionality**; *Formal software verification*; Designing software.

Keywords: QuickCheck, functional programming, formal verification, systems programming

ACM Reference Format:
Zilin Chen, Christine Rizkallah, Liam O'Connor, Partha Susarla, Gerwin Klein, Gernot Heiser, and Gabriele Keller. 2022. Property-Based Testing: Climbing the Stairway to Verification. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3567512.3567520

1 Introduction

Property-based testing (PBT), in the style of QuickCheck [20], is a popular testing methodology and is supported in many modern programming languages [50]. In PBT, tests are specified using logical properties, which are automatically executed on randomly generated inputs in search of counterexamples. PBT is not only useful in finding bugs in programs, it has also been leveraged to reduce the effort in formal verification [15, 29, 37, 47]. Subjecting code to extensive PBT *prior* to verification reduces the number of defects and specification inconsistencies, thus reducing verification cost. Proof engineers can first test a property and only attempt to prove it after having gained reasonable confidence in its validity.

In program verification, it is common practice to prove the correctness of a program against a formal specification. The specification can be given in various forms (e.g. state machines, process calculi, modal logics), depending on the application domain. To show that the implementation conforms to the specification, the notion of *refinement* [7, 24, 52] is frequently used to establish the formal connection.

In this work, we explore the combination of PBT and refinement-based formal verification. We borrow from verification the *functional correctness specification* that is used to dictate the behaviour of the system in question, and give it

SLE '22, December 06–07, 2022, Auckland, New Zealand
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9919-7/22/12…$15.00
https://doi.org/10.1145/3567512.3567520

to PBT. Instead of testing logical properties about the system, which is what PBT is typically designed for, we test the refinement relation between the implementation and the specification. Using logical properties to describe the behaviour of a system has been criticised for its practicability [43], especially if the full functional correctness of the system is desired. Instead, high-level properties of a system can be proved on top of its functional specification.

We introduce PBT to our development cycle, parallel to the refinement-based verification framework. Specifically, we formulate the refinement between the implementation and the functional specification as the property to be tested, which is an under-explored application of PBT. Employing PBT in the formal verification context brings additional benefits beyond detecting bugs in the implementation of the systems.

In contrast to the high-effort all-or-nothing of a full functional correctness proof, PBT provides a continuum ranging from no assurance (no tests), to some assurance (good test coverage), to better assurance (some properties proved, some tested), and all the way to high assurance (all properties proved). This allows users to make trade-offs between cost and assurance according to the criticality of a component.

Tests are more immune to program evolution than formal proofs. A proof may require significant changes whenever the code changes, even in scenarios where the specification remains the same (e.g. optimisation). On the contrary, PBT only requires developers' input when the specification changes. Therefore, PBT can provide quicker feedback on the correctness of the change, reducing code maintenance cost. Furthermore, all proofs depend on assumptions such as the correctness of the hardware or the external software involved. Some of these assumptions can be tested to increase the correctness of the overall system.

It is usually challenging to verify software that was not designed for verification, as the code has to be structured in a modular fashion, around clearly stated correctness properties. This means that it is vital to have an effective means for developers to express their design requirements in order to experiment with and evaluate their design, and to have a good set of design guidelines [14] for them to write programs that can be readily specified and verified.

In large-scale software verification projects, such as seL4 [41], systems developers and verification experts are typically from two separate teams. We posit that PBT specifications can be used to enhance the communication between these two groups. While the properties are similar to formal specifications, they represent tests and, as such, feel more familiar to software engineers than abstract proof requirements. Since PBT gives almost immediate benefit to software engineers, there is an incentive for them to design their code such that these properties are meaningful and easy to express, thereby structuring their code for formal specification, making it amenable to verification.

We examine these benefits by integrating PBT into the Cogent framework. The BilbyFs file system [3], developed in Cogent, provides an example of this effect. The entire BilbyFs had been formally specified but only partially verified. By applying PBT, we uncovered bugs in the specification and the implementation of BilbyFs. PBT has therefore already reduced the cost of verifying the remainder of the system by uncovering mistakes early on.

This work builds on top of a preliminary investigation [18]. To summarise, we make the following contributions:

- We demonstrate how to integrate PBT into a refinement verification framework by using Cogent as the target platform. Unlike previous use of PBT, our test specification is defined in terms of refinement properties (Section 3).
- We argue why PBT is suitable to be employed in parallel with formal verification, and explain the important role that PBT plays in the design and implementation of the systems (Section 4).
- We provide two concrete examples from the testing of components of the BilbyFs file system to demonstrate techniques that we used for specifying refinement relations, modularising the tests, using mocks, handling non-determinism, and efficiently generating test data (Section 5 and Section 6).
- We discuss the engineering implications of our approach and lessons learnt and proposals resulting from them (Section 7).

2 Background

2.1 Cogent

From the experience of formally verifying the seL4 microkernel, Klein et al. [41] have observed that the proofs connecting the high-level specifications with the low-level C systems code are time consuming and tedious to develop, but are not particularly involved. As such they are good candidates for automation. The Cogent verification framework was conceived to partly automate these proofs.

Cogent [55, 56, 58] is a purely functional language that was developed to reduce the cost of developing high-assurance systems components. Similar to the Rust language [39], Cogent is equipped with a uniqueness type system [8, 11, 25, 66] that ensures memory safety, easing the burden of verification. Cogent's type system allows imperative-style destructive updates, while retaining a purely functional semantics. The type system eliminates the need for a garbage collector, making the language more suitable for systems code and also easier to verify.

The uniqueness types come at a cost: it is impossible in Cogent to implement data structures and functions which, even temporarily, rely on sharing. Instead, they have to be implemented in C, verified separately, and imported as *abstract types* and *abstract functions* through a foreign function

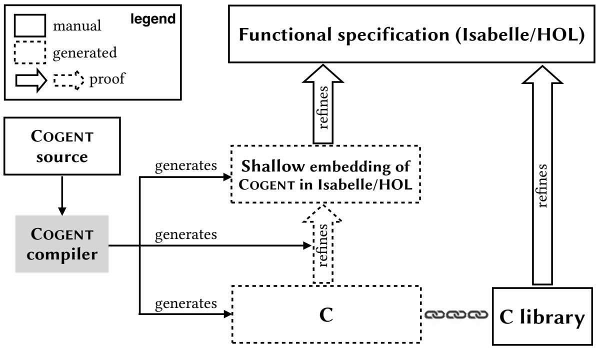

Figure 1. An overview of COGENT's verification framework

interface (FFI) that requires the uniqueness constraints to be satisfied at the interface level. Besides, COGENT does not natively support loops or recursion; they also need to be implemented in C. We refer to this collection of C code as the *C library*.

COGENT's certifying compiler [58, 59], presented in Figure 1, generates C code, a shallow embedding of the COGENT code in the interactive theorem prover Isabelle/HOL [54], and a proof connecting the two. The generated proof ensures that the C code correctly refines the Isabelle/HOL shallow embedding. As such any correctness properties proved about the shallow embedding also hold for the C code. Automating the refinement proof from COGENT to C drastically reduced the verification effort required compared to directly verifying C code [4].

While the COGENT compiler generates C refinement proofs automatically, it cannot prove full functional correctness, which is specified manually by the developer in a *functional correctness specification* in Isabelle/HOL. This leaves two steps that require manual verification (the two solid arrows in Figure 1): (1) verifying that the purely functional shallow embedding of the COGENT code refines the overall functional correctness specification, and (2) verifying that each C ADT refines its functional specification. The former, albeit manual, is eased by virtue of *equational reasoning*. The latter proof, which is directly on the C level, is more involved. However, the C library is to be shared across multiple systems and hence the effort is amortised over time.

Figure 2 provides an example of a COGENT program. Given an abstract `List` datatype with a `reduce` function (line 15) that aggregates the `List` content using a provided aggregation function and identity element, the function `average` (line 17) computes the average of a list of 32-bit unsigned integers. It accomplishes this by storing the running total and count in a heap-allocated data structure, called a `Bag`, defined on line 2, with external allocation and free functions on lines 3 and 4. Because COGENT is a purely functional language, intrinsically impure functions, like the memory (de-)allocation functions, need to have the heap `H` threaded through them. This is similar to Haskell's `IO` monad. The `newBag` function returns a variant (or sum) type to indicate the possibility of allocation failure. The `addToBag` function, which adds a new data point to the bag, is defined on lines 6–8. The `averageBag` function (lines 10–12) returns, if possible, the average of the numbers added to the `Bag`. The input type `Bag!` indicates that the input is a read-only, non-unique view of a `Bag`, which is created on line 21 using the `!` notation. Lastly, lines 17–24 define the overall `average` function, which uses the `Bag` to compute the average of the elements of a `List`.

```
 1 type H
 2 type Bag = { count : U32, sum : U32 }
 3 newBag : H ⟶ <Success (Bag, H) | Failure H >
 4 freeBag : (H, Bag) ⟶ H
 5
 6 addToBag : (U32, Bag) ⟶ Bag
 7 addToBag (x, b { count = c, sum = s })
 8   = b { count = c + 1, sum = s + x }
 9
10 averageBag : Bag! ⟶ <Success U32 | EmptyBag >
11 averageBag b = if b.count == 0 then EmptyBag
12                else Success (b.sum / b.count)
13
14 type List a
15 reduce : ∀ (a, b). ((List a)!, (a!,b) ⟶ b, b) ⟶ b
16
17 average : (H, (List U32)!) ⟶ (H, U32)
18 average (h, ls) = newBag h
19   | Success (b, h') ⟶
20       let b' = reduce (ls, addToBag, b)
21        in averageBag b' !b'
22           | Success n ⟶ (freeBag (h', b'), n)
23           | EmptyBag  ⟶ (freeBag (h', b'), 0)
24   | Failure h' ⟶ (h',0)
```

Figure 2. A simple example of a COGENT program

2.2 Property-Based Testing and QuickCheck

PBT is a quick and effective method for detecting bugs and finding inconsistencies in specifications [38]. Similar to formal verification, PBT uses logical predicates to specify the desired behaviour of functions, by defining the allowed relations between inputs and outputs of the functions. It evaluates the properties on a large set of automatically generated input values in search of counter-examples.

While PBT is effective, it is not universally applicable—in practice, it is often hard to describe the full behaviour of a system solely in terms of logical properties [43]. In the context of formal verification, however, using a functional specification to describe the behaviour of a systems is very common. Thus proof engineers can first run extensive tests on the conjectures before attempting any proof development. This technique is not new and has witnessed great success in the verification community [10].

QuickCheck [20] is a combinator library in Haskell for PBT. While the QuickCheck functionality is now available in many programming languages [50] and theorem provers [15, 29, 47], we interface Cogent to the Haskell QuickCheck library, as it is mature, feature-rich and integrates well with Cogent and C.

2.3 Data Refinement

Prior verification work in Cogent, e.g. of BilbyFs [3], connects the functional specification to the Cogent implementation, and the Cogent implementation to the compiled C code [59] by proving *refinement relation*s. The notation of refinement is also central to our testing framework, in which they are expressed as QuickCheck properties. We use a textbook definition of refinement [24]. Informally, 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.

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. 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.

This state-based model for specifying the behaviours of systems is a very common paradigm in the world of model-based testing (MBT) [33, 43, 64]. However, as mentioned in Section 1, Cogent's purely functional semantics provides a simple formal model of a program's behaviour; specifically, it enables reasoning about programs using *equational* principles. The equational semantics is fortunately widely available in PBT libraries, including QuickCheck.

Since Cogent is a purely functional, deterministic, total language, there is no global state, and all functions are modelled 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 abs :: $X_a \rightarrow Y_a$, and a concrete Cogent function conc :: $X_c \rightarrow Y_c$, 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 abstract specification need not be. In fact, it is often desirable to allow non-determinism to reduce the complexity of the abstract specification. In the context of testing, we are required to restrain the degree of non-determinism in the specification, for the sake of efficient execution of the test script. This, however, does not preclude us from having a non-deterministic specification.

We model non-determinism by allowing abstract functions to return a *set* of possible results. Then, our refinement statement merely requires the single concrete result to correspond to one of the possible abstract results:

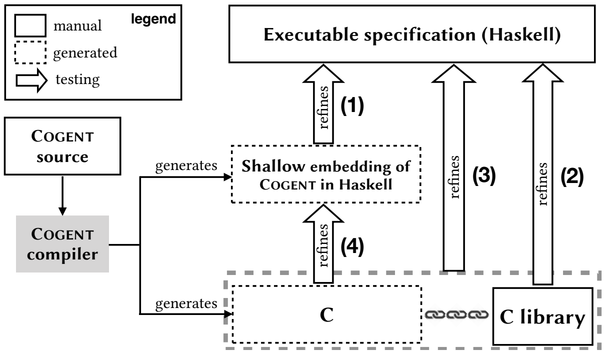

Figure 3. An overview of the Cogent QuickCheck framework

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

Defining the notation

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

the refinement statement can be formulated as:

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

Theorems that capture the functional correctness of Cogent systems typically have this *corres* format. We therefore aim to encode these as machine-testable properties in QuickCheck.

3 The Cogent QuickCheck Framework

The integration of PBT into the Cogent framework mirrors the verification tasks, as shown in Figure 3. The developer manually writes a *Haskell executable specification*, which plays a similar role to the Isabelle/HOL functional correctness specification. The compiler now generates a *Haskell shallow embedding* of the Cogent code for PBT. Although not formally connected, the Haskell and Isabelle/HOL embeddings are very similar.

The framework supports testing the C implementation of ADTs against the Haskell executable specification, shown as arrow (2) in Figure 3; Section 5 provides an example. It furthermore supports testing the Cogent program along with ADTs that the Cogent program uses, against the Haskell executable specification. This is depicted as arrow (1) in Figure 3 (the included ADTs are not shown in the figure); Section 6 provides a case-study. The included ADTs can either be real C implementations (via Haskell's C FFI) or Haskell mocks, depending on whether the ADTs are also considered the system under test.

The refinement relation between the C code and the Haskell embedding of the Cogent program (arrow (4)) can also be tested, although it does not concern us as much, since it is certified by the automatic proof. One scenario where this test can be beneficial is during the development of the

Cogent compiler, before the automatic refinement proof pipeline is fully restored.

The complete compiled executable, depicted in Figure 3 as the grey dotted box at the bottom, can be tested against the executable specification in theory, as indicated by arrow (3). However, we typically do not perform this test using the QuickCheck framework in Cogent, as the gap between their state spaces is usually too large to handle effectively. The final system can normally be deployed in the production environment and be tested against third-party test suites for their specific application domain. In the context of the BilbyFs, for example, there exist tools such as `fstest` [9].

On the formal verification end, the Isabelle/HOL functional correctness specification may be highly non-deterministic in order to succinctly characterise external factors such as the elapsed time of an operation, out-of-memory errors or hardware errors. The Haskell specification must also be capable of modelling non-determinism, but in a more controlled manner, as the specification must be *executable*. Simulating a non-deterministic model can be exponential in time and space. To allow modelling a minimal amount of non-determinism in the Haskell specification, the tester has to ensure that the search domain is finite and reasonably small by carefully examining the needed quantifiers in the specification.

For example, in the Isabelle/HOL abstract specification of BilbyFs, the `afs_get_current_time` function is defined as follows:

```
definition
  afs_get_current_time :: afs_state ⇒ (afs_state ×
     TimeT) cogent_monad
where
  afs_get_current_time afs ≡ do
    time′ ⟵ return (a_current_time afs);
    time ⟵ select {x. x ≥ time′};
    return (afs⦇ a_current_time := time ⦈, time′)
  od
```

It non-deterministically picks a `time`, which is no earlier than the time stored in the file system state `afs`. This abstract specification is not suitable for testing, due to the infinitely large set of values. In contrast, on line 13 of Figure 5b, the specification non-deterministically chooses an error code from a small set of `{eIO, eNoMem, eInval, eBadF, eNoEnt}`. This moderate state explosion can potentially be handled by the testing framework, depending on the context in which the `afs_readpage` function is applied.

The Haskell executable specification is thus often strictly less abstract than the Isabelle/HOL functional specification. Although it is not necessary to formally connect the two, as the gap only affects the quality of the tests, ideally we would want one specification to be generated from the other one. Automatic refinement mechanisms that allow verified generation of Haskell from Isabelle/HOL have been explored [44, 45]. Generating the Haskell specification from the Isabelle/HOL abstract specification is undoubtedly handy, when the Isabelle/HOL specification is developed prior to the Haskell executable specification. This however is not always the case, and in fact in the workflow that we proposed, the Haskell specification is used to guide the formulation of the Isabelle/HOL one.

4 PBT and Systems Design Go Hand-in-Hand

We argue that the employment of PBT and the design of the systems are interlinked with each other: appropriate systems design assures the effectiveness of testing, and the use of PBT encourages the programmers to design their systems in a fashion that is amenable to formal verification.

As a purely functional language, Cogent is well suited for PBT and verification: the result of a executing a function only depends on the input, with no hidden state. Instead, a system state must be explicitly threaded through an impure function. If the function is not designed appropriately, it is possible that a PBT test suite hardly yields counter-examples. Systems code often involves large global states. However, a function typically only accesses a small portion of the state. If the entire state is passed in, any random variations to the parts of the state that are irrelevant to the function will have no effect on the execution of the function. In this case a large proportion of the randomly generated test cases in PBT will be wasted, rendering the test suite ineffective.

In practice, this means that to be suitable for PBT, the functions must be designed to keep the inputs minimal and relevant, which is unlikely when naïvely translating existing C code with global state into Cogent. While this may seem like a high price to pay, a verification-friendly design has the same requirement for modularity and compartmentalisation [3, 5]. Thus PBT imposes few restrictions beyond existing requirements of verification, and instead helps guide the design.

In the context of Cogent, developers typically do not have a formal specification to begin with when implementing a new system. Systems programmers, together with verification experts, not only need to ensure that they are implementing the systems right, but also to ensure that they are implementing the right systems. PBT helps them structure the implementation, as well as the specification. Having good design decisions, such as keeping the states threaded through small and relevant as we mentioned above, is doubly rewarding: it keeps the refinement relation between the concrete and the abstract states simple, both in PBT and in verification.

Expressing design requirements for verification is an ongoing challenge, and we observed in the past that when verifying real-world systems, it is difficult for proof engineers to communicate these requirements effectively to the software engineers. The seL4 project [41] overcame this problem by using an executable Haskell specification of the system as

```
prop_corres_wordarray_set_u8 :: Property
prop_corres_wordarray_set_u8 = monadicIO $
  forAllM gen_wordarray_set_u8_arg $ \args ⟶ run $ do
    let ia = mk_hs_wordarray_set_u8_arg args
        oa = uncurry4 hs_wordarray_set ia
    ic ⟵ mk_c_wordarray_set_u8_arg args
    oc ⟵ cogent_wordarray_set_u8 ic
    corresM' rel_wordarray_u8 oa oc
```

Figure 4. The refinement statement for `wordarray_set` (deallocation is omitted for simplicity)

an interface between these two groups [40]. We posit that QuickCheck properties is a highly suitable language for communicating design requirements. They readily translate to formal specifications, but are expressed in a programming language, and thus familiar to software developers. As they lead to effective test generation, software developers get immediate benefit from using these specifications to structure their code to maximize their use, which consequently makes the code easier to verify.

5 Example: The `WordArray` Library

We apply the Cogent QuickCheck framework to the `WordArray` library, which implements common functions manipulating arrays of machine words and is shared by all of our systems implementations. Most of these `WordArray` functions are implemented in C, and are invoked via the FFI mechanism available in Cogent.

We want to test whether each function observes the refinement property from Section 2.3. For example, the behaviour of the ADT function `wordarray_set` (similar to `memset` in C)—which fills the first `n` elements starting at a certain index `frm` into an array `arr` with a constant value `a`—is manually specified in Haskell as follows:

```
-- Haskell spec.
type WordArray a = [a]
hs_wordarray_set :: WordArray a ⟶ Word32 ⟶ Word32
                 ⟶ a ⟶ WordArray a
hs_wordarray_set arr frm n a =
  let len = length arr in
   if | frm > len = arr
      | frm + n > len
      = take frm arr ++ replicate (len - frm) a
      | otherwise
      = take frm arr ++ replicate n a ++
        drop (frm + n) arr
```

In Cogent, the function is defined as an abstract function, whose definition is given in C. The Cogent function interface looks like:

```
type WordArray a
wordarray_set : (WordArray a, U32, U32, a) ⟶ WordArray a
```

While Cogent and Haskell both support polymorphism, C does not, and QuickCheck cannot perform genuine polymorphic testing [12]. In this example, we test the `U8` instance of the polymorphic `wordarray_set` function, whose refinement statement is given in Figure 4. It can be read roughly as: for any type-correct concrete input `ic` and its abstraction `ia`, check that the result (i.e. `oc`) of applying the concrete function `cogent_wordarray_set_u8` and that of the abstract function `hs_wordarray_set` (i.e. `oa`) are related via the refinement relation `rel_wordarray_u8`. The `corresM'` function is a monadic variant of our *corres* notation for situations where the specification is deterministic.

Although the *corres* predicate contains an existential quantifier for the result of the non-deterministic abstract specification, our implementation does not require QuickCheck to guess the quantified value from a set of all possible values. Instead, our test driver enumerates over all possible output values of the abstract function to find the existentially quantified value. In Section 6, we show how to restrict the size of the abstract function's codomain for the enumeration to be tractable.

For the `WordArray` type, we relate the abstract input data and the concrete input data in the following way: we randomly generate test data on a middle-ground type, and then use two thin wrappers `mk_hs_wordarray_set_u8_arg` and `mk_c_wordarray_set_u8_arg` to convert the generated data to the types expected by the abstract and the concrete functions. The test data generation is not very involved, because the correspondence between the two types is straightforward, and the C type is not very convoluted in its underlying representation, in particular it does not heavily use pointers.

Although in the refinement statement, the refinement relation between the input data is expressed as a predicate, this is usually not the way to implement the test driver. Checking a predicate is often straightforward, but it requires two sets of random data generators and forces the two generators to be coupled. Otherwise the correspondence predicate is likely to reject the vast majority of the generated data, rendering the test very inefficient (see Section 7.4).

Broadly speaking, it is more convenient to relate the input data if we implement the refinement relation as an *abstraction function*, computing the abstract data according to its concrete counterpart. This is contrary to model-based testing approaches [64], in which the test cases are derived from the more abstract model. We use abstraction functions because typically the concrete state contains more data than its abstract counterpart. In order to derive a concrete input from the randomly generated abstract input, more data need to be created and this calls for another set of random data generators. On the other hand, when we generate a concrete input and abstract it, it only requires a lossy abstraction function.

In the case of `WordArray`, we compute the abstract input data from the concrete one, relating them by construction. However, not all values of a C type are valid inputs: for instance, a null pointer does not correspond to a valid

WordArray. To exclude invalid input data, we manually implement a test data generator gen_wordarray_set_u8_args which generates values that are isomorphic to valid concrete inputs only, and we convert them to Haskell inputs and C inputs using functions mk_hs_wordarray_set_u8_arg and mk_c_wordarray_set_u8_arg respectively.

The refinement statement as shown in Figure 4 is largely boilerplate code. To generate this code, we designed a small domain-specific language (DSL), whose prototype has been implemented [27]. In this DSL, programmers can specify the function names, the definition of the abstraction function for the inputs and the refinement relation between the outputs, and other properties about the refinement statement, such as the determinism of the abstract function and whether the concrete function needs to operate under the IO monad. The DSL is written in JSON format, which can be readily parsed using third-party libraries such as aeson [57]. A piece of sample code is give below:

```
1 {
2   "name"  : "wordarray_set_u8",
3   "monad" : true,
4   "nondet": false,
5   "absf"  : ... // the abstraction function
6   "rrel"  : ... // ref. rel. between outputs
7 }
```

Haskell program texts can be embedded in the JSON structure as the values of the "absf" and "rrel" attributes (lines 5 and 6). This allows programmers to either call a Haskell function defined elsewhere, or directly write the definition in-place. The lens [42] style of code is particularly suitable for accessing and relating parts of deeply nested algebraic datatypes.

In the refinement statement, the Cogent-compiled C code can be called from Haskell using its C FFI facility. The Haskell representation of C types, marshalling functions, and foreign function calls are generated by the Cogent compiler and are further compiled by FFI tools such as hsc2hs [32] and c2hs [16].

Running a small number of randomly generated tests (by default 100 but this can be customised) by passing prop_corres_wordarray_set_u8 to the quickCheck function, we get:

```
*WordArray> quickCheck prop_corres_wordarray_set_u8
+++ OK, passed 100 tests.
```

We have specified most of the ADT functions for WordArrays and tested them [17]. We found bugs in two C functions, which had not been uncovered by our earlier test suites nor the file systems built with them. The bugs went undetected as they involved invalid inputs and corner cases which were handled by the callers, whereas the Haskell specification in our QuickCheck framework does not preclude these input values.

For example, for the wordarray_copy function that copies a number of bytes from one memory area to another (similar to memcpy in C), the old implementation implicitly assumed that the index into the source array was always within bounds. This precondition was satisfied by our file system implementations, but it was unspecified. In fact, the wordarray_copy function, as part of a generic library, should not carry this implicit precondition. Otherwise it may introduce bugs to other customers of this library function but do not perform the check.

PBT also helped us uncover problems in the Isabelle/HOL ADT specifications, which had overly specific assumptions about inputs. While these assumptions are valid for the functions we verified, they do not hold in general. Thus, the specifications we had written did not represent a general purpose specification of the function.

The WordArray library in Cogent was initially axiomatised in the verification of the file systems [4], and then tested using the PBT framework, before they were finally formally verified [19]. This is an example of how the developers can progressively increase their confidence in the correctness of the code by upgrading PBT to formal verification in a modular fashion.

6 Example: A Top-Level File System Operation

BilbyFs [3] is a flash file system that was designed from scratch, focusing on modularity and verifiability; it has 19 top-level file system operations. Two functions have been previously verified in Isabelle/HOL to demonstrate how Cogent facilitates equational reasoning. fsop_sync, a top-level function, consists of about 300 lines of Cogent code and took approximately 3.75 person months to verify with 5700 lines of proof. The other function, iget, directly called by the top-level fsop_lookup function, consists of approximately 200 lines of code, and took about one person month to verify with 1800 lines of proof.

We conducted PBT on one of BilbyFs's top-level function fsop_readpage [17], which had previously been formally specified in Isabelle/HOL but not yet verified. Figure 5 shows the Isabelle/HOL specification as well as the manually written Haskell executable specification; they are very similar in this case. Therefore, testing gives us reasonably high assurance of the implementation with respect to the Isabelle/HOL specification. As discussed in Section 3 , this is not always the case, making it occasionally more difficult to connect the two specifications, and sometimes requiring additional manual reasoning.

6.1 The Haskell Executable Specification

In a nutshell, as shown in the Haskell specification in Figure 5a, the function hs_fsop_readpage fetches a designated data block of a specific file to the buffer. The argument afs is a map from inode numbers to files; each file is represented as a list of blocks of data. The hs_fsop_readpage function looks

```
hs_fsop_readpage :: AfsState
                 → VfsInode
                 → OSPageOffset
                 → WordArray U8
                 → NonDet (Either ErrCode (WordArray U8))
hs_fsop_readpage afs vnode n buf =
  let size = vfs_inode_get_size vnode :: U64
      limit = size `shiftR` bilbyFsBlockShift
   in if | n > limit → return $ Left eNoEnt
         | n == limit && (size `mod` bilbyFsBlockSize == 0) →
             return $ Right buf
         | otherwise → return (Right $ fromJust (M.lookup (vfs_inode_get_ino inode) afs) !! n) <|>
                        (Left <$> [eIO, eNoMem, eInval, eBadF, eNoEnt])
```

(a) The Haskell executable specification

```
definition
  afs_readpage :: afs_state
               ⟹ vnode
               ⟹ U64
               ⟹ U8 WordArray
               ⟹ (U8 WordArray × (unit, ErrCode) R) cogent_monad
where
  afs_readpage afs vnode n buf ≡
   if n > (v_size vnode >> unat bilbyFsBlockShift) then
    return (WordArrayT.make (replicate (unat bilbyFsBlockSize) 0), Error eNoEnt)
   else if (n = (v_size vnode >> unat bilbyFsBlockShift)) ∧ ((v_size vnode) mod (ucast bilbyFsBlockSize) = 0)
        then return (buf, Success ())
        else do err ⟵ {eIO, eNoMem, eInval, eBadF, eNoEnt};
                return (WordArray.make (pad_block ((i_data (the $ updated_afs afs (v_ino vnode))) ! unat n)
     bilbyFsBlockSize), Success ()) ⊓
                return (buf, Error err)
             od
```

(b) The Isabelle/HOL functional specification

Figure 5. Functional specifications of the `fsop_readpage` function

up a file, whose inode number is given by `vnode`, in the map `afs`, and copies the `n`-th block of the file to `buf`. It returns non-deterministically an updated buffer or an error code.

As a first step, `hs_fsop_readpage` calculates the number of blocks that the wanted file occupies. If the block in question is out of bounds (`n > limit`), the function returns a no-entry error `eNoEnt`. If the file size is a multiple of the block size, `n` points to the last block in the wanted file, and the last block is empty (because the file data ends at the prior block boundary), then the function returns the original buffer as there is no data to read. Otherwise, `hs_fsop_readpage` reads the block by looking up the inode number in the map (see Figure 6 for a pictorial example).

This, however, is not the only possible correct behaviour. As the implementation has to access buffers and read from the physical medium, this may fail, in which case it should throw an error. We specify this as a non-deterministic behaviour. The specification states that the function can read a block or it can give one of the following five errors: `eIO`, `eNoMem`, `eInval`, `eBadF`, or `eNoEnt`. The `NonDet` monad used here is essentially a finite set containing all allowed behaviours. This monad is commonly used in proving refinement (e.g. [3, 22]). The alternative operator `(<|>)` acts as a non-deterministic choice, admitting the behaviour of either of its operands by taking the union of their behaviours.

6.2 Mock Implementations

It is not always feasible to test systems code in its exact production environment [53]. For instance, the `fsop_readpage` example has many low-level functions which call into the operating system's kernel, and it is currently not feasible to run QuickCheck tests in kernel mode. Instead of testing the monolithic object file obtained from compiling the C code, we mock up parts of the code in Haskell. A mock abstracts from low-level kernel calls and can be thought of as a *black box*, which provides to its caller the same observable effects as the actual implementation.

Mocks can also be used as substitutes for unimplemented functions, enabling systems developers to test functionality before they have a full system implementation. The use of

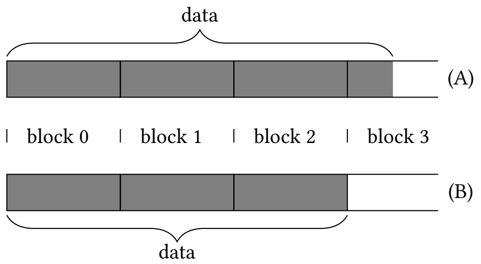

Figure 6. An example of the `read_page` algorithm. In case (A), `limit` = 3. When `n` = 0, 1, 2 we just read. When `n` = 3, because the size of the data is not perfectly aligned at the end, we still read. When `n` ≥ 4, we return the no-entry error. In case (B), `limit` = 3. When `n` = 3, that's the special case. We return the old buffer unmodified.

mocks restricts the scope of debugging to a small number of functions, reducing the effort required to locate bugs.

The Cogent implementation of `fsop_readpage` calls a `read_block` function to fetch one block of file data, which in turn retrieves the data with the function `ostore_read`. The `read_block` function, which retrieves the file data from the physical medium, is conceptually simple. However it is complicated in its internal implementation, which involves a red-black tree lookup to locate the address, several layers of caching, and thorough error-handling.

Because `read_block` relies on kernel mode access to caches and complex data structures, it is a good candidate for a mock implementation. In addition to the `ostore_read` function, we also substitute some kernel ADT functions for mock implementations. For `WordArray` functions invoked by `fsop_readpage`, we use the Haskell models described in Section 5 as mocks.

The implementation of a mock is simplified by the fact that it does not need to provide the full functionality of the operation, as long as the callers in the specific test cases cannot observe the difference in its behaviours.

For example, the original Cogent signature of the function `ostore_read` is:

```
type RR x a e = (x, <Success a | Error e>)
ostore_read : (SysState, MountState!, OstoreState, ObjId)
            → RR (SysState, OstoreState) Obj ErrCode
```

The function takes a quadruple as input, containing a read-only (denoted by the `!` operator) `MountState`, and returns the parametric `RR` type, which is further defined as a pair of a variant type `<Success a | Error e>` and a result type `x` that is common to both cases.

The purely functional nature of Cogent makes it easy for developers to identify the observable behaviours—they are necessarily within the return type of a function. In the case of the `fsop_readpage` function, the only behaviours of `ostore_read` that can be observed are the returned `Obj` value or the error code `ErrCode`. Therefore, we can tailor the mock to the specific use case of `fsop_readpage`. The mock `ostore_read` function can be modelled as a simple map lookup: given a map `OstoreState` and a key of type `ObjId`, it returns the corresponding object or an error. The relevant Haskell definitions are given as follows (the use of the `Oracle` can be ignored for now; it will be explained shortly):

```
type OstoreState = Map ObjId Obj
data Res a e = Success a | Error e

ostore_read :: Oracle
   → (OstoreState, ObjId) → Res Obj ErrCode
ostore_read orc (ostore_st, oid) =
  if orc == 0 then
    case M.lookup oid ostore_st of
      Nothing  → Error eNoEnt
      Just obj → Success obj
  else Error orc
```

6.3 Oracles and Non-determinism

The Cogent implementation of `ostore_read` interacts with the physical media and kernel data structures, therefore its behaviour is dependent on that of the underlying systems and hardware. However, as we have introduced in Section 2.1, Cogent is a total, purely functional language, meaning that all functions in Cogent have to be deterministic. This non-determinism, therefore, has to be modelled by threading a global state `SysState` through the impure functions, similar to the `H` type in Figure 2.

In the Haskell mock implementation of `ostore_read`, the non-determinism can be modelled in a different manner for simplicity. When testing this function independently, we emulate the non-determinism by adding an additional *oracle* input `orc`.[1] This oracle can be deemed to be the source of all non-determinism. A similar oracle is passed to our mocks of many `WordArray` functions, such as `wordarray_create`, which allocates memory and creates a fresh array.

We have seen how oracles can be used in mock implementations to emulate non-determinism. The oracle technique can be similarly applied to the Haskell executable specification. When we test that an oracle-carrying mock refines a non-deterministic specification, the specification can abstract over the values that the oracle can possibly possess with the `NonDet` monad. If the specification is to be made more precise, we can also introduce an oracle to it to ascribe the source of the non-determinism. In this case, care needs to be taken to ensure that the oracle in the specification and that in the implementation are in synchronisation, so that they do not make conflicting choices.

Using oracles in the specification can be problematic if the implementation is not a mock and is genuinely non-deterministic due to, say, the hardware or the operating system. There is in general no way to predict accurately which

[1] In our implementation, we pass the oracle around using GHC Haskell's implicit parameter extension [48], making it more transparent to users. In the presentation of this paper, however, we pass them explicitly for clarity.

execution path the concrete implementation will take (e.g. `malloc` failures). Hence, the specification and the implementation can make inconsistent choices when they encounter non-determinism, which can lead to spurious test failures. In this case, we have to step back and use a non-deterministic specification with the `NonDet` monad instead. This, however, has a negative cosmetic effect on the entire Haskell specification, as the `NonDet` monad is infectious and it would render all ancestor functions in the call graph monadic.

To address this problem, we can establish an equivalence relation between the non-deterministic specification using `NonDet` and the oracle-carrying deterministic one by testing. Concretely, we test that the two specifications return the same set of results, by enumerating every possible oracle in the deterministic specification, collecting a finite set of results, and then checking that set against the set of results produced by the non-deterministic specification.

The `NonDet` monad and the oracle approach are two extremes of the spectrum, and developers can choose a suitable degree of non-determinism by combining these two to meet the needs of a specific test case.

6.4 Test Data Generation

By using mocks, not only can we use simpler algorithms to simulate the functionality of the original components, but we can also fine-tune test data generators to restrict the domain of inputs given to the mock, allowing us to only implement a partial mock of the original code.

When testing a cluster of functions and the mock function's input depends on the output of other functions, the aforementioned partial mock should be used with care. The input to a function can be directly controlled by the tester by defining appropriate test data generators, while the output of a function is not so easy to predict as it may have been heavily processed and manipulated by the functions. When such an input value reaches the partial mock implementation, it is harder to know a priori whether it falls into the unhandled sub-domain of the mock. It poses a greater challenge in writing good test data generators to ensure the pre-condition of the mock function is met even after the randomly generated data have flowed through other functions in the system under test. More general remarks on this point can be found in Section 7.4.

Domain-specific knowledge can be leveraged to write good test data generators, which makes the checking process more efficient and practicable. For example, when we generate the `OstoreState`, all entries we generate belong to the same inode. In reality, there are many data objects for other files, or other types of objects; but none of these facts will be observed by its caller. This in turn simplifies the implementation of the mock and the abstraction functions.

6.5 Results

The shape of the top-level refinement statement for the Haskell shallow embedding of the Cogent `fsop_readpage` function (shown below) closely resembles that of the `WordArray` example. An oracle is also generated and passed to the Haskell embedding, which will be further passed to the mock implementation of `ostore_read` as discussed earlier.

```
prop_corres_fsop_readpage :: Property
prop_corres_fsop_readpage =
  forAll gen_fsop_readpage_arg $ \ic ⟶
    forAll gen_oracle $ \o ⟶
      let ia = abs_fsop_readpage_arg ic
          oa = uncurry4 hs_fsop_readpage ia
          oc = fsop_readpage o ic
       in corres rel_fsop_readpage_ret oa oc
```

From the counter-examples produced by QuickCheck, we found that the Haskell executable specification was flawed, which in turn exposed a problem with the Isabelle/HOL abstract specification, from which the Haskell specification was derived. These specifications did not take errors returned from `ostore_read` into account. Testing the above property helped us rectify this mis-specification.

7 Design Decisions and Key Takeaways

We have showcased two applications of the PBT framework in Cogent. Since the examples we examined are from a real file system, they have given us some good insights into how much boilerplate code is required, which components of the testing infrastructure can be automatically generated, and how these pieces can be integrated.

7.1 Modular Testing and Whole-System Testing

Our QuickCheck machinery does not require the user to test the entire system at once. Instead, the user may test refinement for each function or for a cluster of functions at a time. Typically, the ADTs implemented in C form a common module, shared across many systems. Accordingly, our framework allows developers to test the ADTs in isolation, with no regard to how they are used within systems. This modularity is aided by Cogent's functional semantics.

For the `fsop_readpage` example, we chose to employ modular testing as opposed to whole-system testing, which would have required extra effort in developing the infrastructure to run tests in kernel mode (see Section 7.3). Whole-system testing allows for more abstract top-level properties to be specified: for instance, that read and write are inverse operations.[2] Alternatively, these logical properties can be tested

[2] It might be surprising to some readers that we claim that this property is suitable for whole-system testing, instead of PBT. After all, this kind of round-trip properties are typical in the realm of PBT. The reason is that, systems software, or file systems in this case, are not implemented cleanly in a purely functional manner. They always involve heavy I/O, kernel interaction, locking, etc., which cannot be precisely and concisely specified in the functional specification and thus fall under the global state.

on top of the executable specification rather than directly on the concrete implementation.

As our specification becomes more abstract, the single step simulation style of refinement becomes less relevant, because several low-level functions may be specified as a single function on the abstract level. Modular testing, on the other hand, is more comprehensive, as it also examines the interfaces among different components in a system. In this case, it uncovered issues in the `WordArray` implementation that whole-system testing would have, and indeed had, missed.

7.2 Functional Specification Versus Logical Properties

Traditional PBT tests specifications against a set of logical properties (e.g. *get* and *set* are inverse operations on `WordArrays`). We instead test functions against a full executable specification that models the functions. This is conceptually similar to model-based testing [43] (also see Section 8). The functional specification is most akin to the functional notions of model paradigm as classified in the work by Utting et al. [64, § 3.3].

It is often easier to use a model rather than a set of properties to define the behaviour of functions [43]; our experiments concurred. For example, functions in the C library can be readily modeled in terms of Haskell library functions. Moreover, low-level functions in systems programming, e.g. setting a flag, are often very simple in its functionality, but can hardly be characterised by traditional properties that are abstract and intuitive enough for users to comprehend.

Using functional specifications encourages compositionality. The functional specification of one module can also be used as a mock implementation when testing other modules that depend on this module. For instance, in our `fsop_readpage` case study, we used the previously defined Haskell functional specification of the `WordArray` functions as mocks.

Furthermore, functional specifications serve as a communication interface between system programmers and proof engineers. They are key to designing verification-friendly systems programs, whereas logical properties alone fall short in this aspect.

7.3 Testing Kernel Modules

A file system is typically compiled as a kernel module and runs in kernel mode, while our test framework runs in user mode. To handle this discrepancy, for our prototype, we have ported our file systems code to run in user mode, using mocks to simulate the kernel APIs. Emulating the kernel is common practice in systems programming, with libraries such as FUSE [31] facilitating user-space execution of kernel code. However, these tools expect a complete kernel module, thus precluding the use of mocks or other user-land code during testing. A possible alternative to explore is using a system such as KML [51], House [34] or HaLVM [35] to run PBT in kernel mode. This would allow the testing environment to more closely resemble the real run-time environment of the software. We leave it for future investigation.

7.4 Test Generation Strategies

There are two main factors to consider when generating test data for PBT. The first is how to sample the data: we choose user-guided random test generation à la QuickCheck in this work. Exhaustive testing (for small values) is another popular strategy and has gained great popularity, e.g. SmallCheck for Haskell [60]. However, the small scope hypothesis on which SmallCheck is based does not hold in general in the context of systems software.[3] For instance, integer overflow, which is a common bug in systems code, can hardly be triggered by small values. The second main concern is the effective generation of test data which satisfies the premises of the properties. If a property has the form $p \implies q$ and the premise p is very strong (i.e. difficult to satisfy), and the test data is not sampled with great care, then a lot of them will falsify the premise and thus be discarded in the test, rendering the test inefficient. All refinement statements in this work have the form $p \implies q$ with a strong precondition p. The Luck framework [46] couples the predicates of the property and the test data generation, which could simplify writing custom test data generators. There is a rich body of research devoted to test generation techniques [28]. In the future, we plan to explore more options to automate our test data generators.

7.5 Shrinking

Counter-example shrinking reduces the size of counter-examples before reporting them to testers, which helps developers better understand and fix bugs. The Haskell QuickCheck provides a customisable shrinking library with a default shrinking algorithm for many datatypes. A rich body of research can be found on more advanced shrinking algorithms. For example, test data shrinking that preserve invariants about the generated data has been explored in [49, 62]. But due to the lack of recursively defined datatypes in COGENT and thus in the COGENT-powered file systems, the effectiveness of shrinking is dubious, as the size of the input data chiefly comes from the sheer complexity of the (non-recursive) datatypes, rather than from recursion. Shrinking is nevertheless useful for testing ADTs, but basic shrinking strategies work reasonably well in our context.

8 Related Work

QuickCheck has been used for testing a variety of high-level properties, such as information flow control [23, 37], mutual

[3] The small scope hypothesis is stated in Runciman et al. [60]'s paper as: "(1) If a program fails to meet its specification in some cases, it almost always fails in some simple case. Or in contrapositive form: (2) If a program does not fail in any simple case, it hardly ever fails in any case."

exclusion [21], and the functional correctness of AUTOSAR components [6, 53]. To the best of our knowledge, our framework is the first to use PBT for testing refinement-based functional correctness statements.

The hs-to-coq tool [61] translates Haskell code into the Coq proof assistant [13]. Breitner et al. [14] used it to verify parts of Haskell's container library in Coq. In addition to proving the functional correctness of various functions in the library, they also verified that the QuickCheck properties that the library is tested against are correct. By contrast, our QuickCheck properties are refinement properties that directly resemble the those used for full verification. Verifying these properties is already a substantial step towards proving functional correctness, and in some cases directly implies functional correctness.

QuickCheck is available as a built-in tool in Isabelle/HOL and is used for quickly finding counter-examples to proposed lemmas [10, 15]. We chose to build on Haskell's QuickCheck rather than Isabelle/HOL's QuickCheck because it is easier for Cogent programmers to use a testing framework that lies in the ecosystem of a functional programming language rather than interact with a theorem prover. Haskell acts as a good communication medium between programmers and proof engineers [14, 26]. Moreover, due to Isabelle/HOL's interactive nature, testers would have to wait for Isabelle to re-process the proof scripts affected by a change in a theory file, before they can run tests again. Even if Isabelle's quick-and-dirty mode is enabled, which skips proofs, testers would still have to wait for Isabelle/HOL to process definitions. In fact, a large portion of the time is spent on reading in the deep embeddings of the Cogent program into Isabelle, due to the large terms generated by the Cogent compiler. This would cause a significant and unnecessary reduction to their productivity, and destroy the user experience.

The SPARK language [2], a formally defined subset of Ada, also uses a combination of testing and verification to facilitate the development of high-reliability software. SPARK developers can attach contracts, that is, specifications of pre- and postconditions, to critical procedures. Tools of the framework can use these contracts as input to automatically test the procedures, or attempt to formally prove that the implementation observes these contracts. Ada language features that are hard to verify, such as side-effects in expressions, access types, allocators, exception handling and many others, are not permitted in SPARK. SPARK focuses on selectively verifying safety critical components, rather than fully verified systems from high-level specification to machine code.

DoubleCheck [30] integrates PBT into Dracula [65], a pedagogical programming environment which enables students to develop programs and then prove theorems about them in ACL2 [1], a theorem prover based on term rewriting. As with our work, the motivation of this integration is to facilitate formal verification, though its focus is on education, not on producing verified real-world applications.

In the PBT framework we presented, as we test the refinement statement between the implementation and the Haskell executable specification, which can be considered a *conformance relation*, it does appear that we are instead conducting model-based testing [63, 64]. Our approach does indeed share a lot in common with MBT, but we identify our approach as PBT for the following reasons. Firstly, in MBT, the starting point of testing is a model of the software under test. In contrast, as we have demonstrated, testing in our framework does not necessarily have an existing model to start with. In developing formally verifiable operating systems components, which is the application domain that concerns us, it is of paramount importance to find the right balance between verifiability and performance. PBT gives developers insights in both aspects. Therefore, testing plays a role in the design of the system, and subsequently its specification. This is similar to the iterative development process reported in the seL4 formal verification work [36]. Secondly, test cases are systematically and algorithmically generated from the model in MBT. Test inputs are typically concretised from the abstract test suite and the test results are abstracted to be validated against the model by an adapter. In contrast, as we have shown in the examples, our test cases are not generated from the specification; test data is directly produced on the concrete level. Lastly, from the tooling perspective, our approach uses a PBT library QuickCheck as the core of the testing infrastructure.

9 Conclusion

In this paper, we showed how we augmented the Cogent verification framework with PBT. Testing and formal verification complement each other, which is well acknowledged among researchers and developers. In this work, we further demonstrated this common belief in the specific context of PBT and interactive theorem proving. The central idea is to mirror the refinement proof in testing, using a functional specification as the model instead of a set of logical properties as commonly done in PBT.

Using this method, we tested an abstract data type from a library, as well as an operation of a real-world file system. The tests exposed several bugs in the ADT implementation and uncovered errors in the specification of the ADT and of the file system.

Besides the main purpose of testing—detecting bugs—we exhibited other benefits of employing PBT. It reduces the effort in formal verification, guides the development of verification-ready specifications and programs, and acts as a precise and effective communication media among developers. We believe PBT offers developers the opportunity to gradually tackle the verification challenge in large and complex systems development, serving as a helpful stepping stone in the endeavour into full formal verification of high assurance software.

References

[1] ACL2. 2022. ACL2. Retrieved October 2022 from http://www.cs.utexas.edu/users/moore/acl2/

[2] AdaCore. 2022. SPARK Pro. Retrieved October 2022 from https://www.adacore.com/sparkpro/

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

[4] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 175–188.

[5] 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.

[6] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *International Conference on Software Testing, Verification and Validation (ICST) Workshops*. Graz, AT, 1–4. https://doi.org/10.1109/ICSTW.2015.7107466

[7] R. J. R. Back. 1988. A calculus of refinements for program derivations. *Acta Informatica* 25, 6 (Aug. 1988), 593–624. https://doi.org/10.1007/BF00291051

[8] Erik Barendsen and Sjaak Smetsers. 1993. Conventional and Uniqueness Typing in Graph Rewrite Systems. In *Foundations of Software Technology and Theoretical Computer Science (Lecture Notes in Computer Science, Vol. 761)*. 41–51.

[9] Brian Behlendorf. 2011. *POSIX Filesystem Test Suite.* Retrieved August 2022 from https://github.com/zfsonlinux/fstest

[10] Stefan Berghofer and Tobias Nipkow. 2004. Random Testing in Isabelle/HOL. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference (SEFM '04)*. IEEE Computer Society, Washington, DC, USA, 230–239. http://dx.doi.org/10.1109/SEFM.2004.36

[11] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL (Dec. 2017), 5:1–5:29. http://doi.acm.org/10.1145/3158093

[12] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. 2010. Testing Polymorphic Properties. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 125–144.

[13] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer.

[14] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, set, verify! applying hs-to-coq to real-world Haskell code (experience report). *PACMPL* 2, ICFP (2018), 89:1–89:16.

[15] Lukas Bulwahn. 2012. The New Quickcheck for Isabelle: Random, Exhaustive and Symbolic Testing Under One Roof. In *International Conference on Certified Programs and Proofs*. Springer-Verlag, Berlin, Heidelberg, 92–108. http://dx.doi.org/10.1007/978-3-642-35308-6_10

[16] 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

[17] Zilin Chen. 2022. Cogent property-based testing case studies. https://github.com/au-ts/cogent/tree/master/impl/fs/bilby/quickcheck.

[18] Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. 2017. The Cogent Case for Property-Based Testing. In *Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, Shanghai, China, 1–7.

[19] Louis Cheung, Liam O'Connor, and Christine Rizkallah. 2022. Overcoming Restraint: Composing Verification of Foreign Functions with Cogent. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2022)*. ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/3497775.3503686

[20] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th International Conference on Functional Programming*. 268–279.

[21] 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 *International Conference on Functional Programming*. ACM, New York, NY, USA, 149–160. http://doi.acm.org/10.1145/1596550.1596574

[22] David Cock, Gerwin Klein, and Thomas Sewell. 2008. Secure Microkernels, State Monads and Scalable Refinement. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*. Springer, Montreal, Canada, 167–182.

[23] 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 *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, CA, USA, 165–178.

[24] Willem-Paul de Roever and Kai Engelhardt. 1998. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, United Kingdom.

[25] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2008. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages (Lecture Notes in Computer Science, Vol. 5083)*. Springer, 201–218.

[26] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. 2006. Running the Manual: An Approach to High-Assurance Microkernel Development. In *Proceedings of the ACM SIGPLAN Haskell Workshop*. Portland, OR, USA.

[27] Oscar Downing. 2021. *Enhancements to the Cogent Property-Based Testing Framework.* Undergraduate Thesis. CSE, UNSW, Sydney, Australia. https://people.eng.unimelb.edu.au/rizkallahc/theses/oscar-downing-honours-thesis.pdf

[28] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 61–72. http://doi.acm.org/10.1145/2364506.2364515

[29] Peter Dybjer, Haiyan Qiao, and Makoto Takeyama. 2003. Combining Testing and Proving in Dependent Type Theory. In *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 188–203.

[30] Carl Eastlund. 2009. DoubleCheck Your Theorems. In *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '09)*. ACM, New York, NY, USA, 42–46. http://doi.acm.org/10.1145/1637837.1637844

[31] FUSE. 2022. The FUSE Project. Retrieved October 2022 from https://github.com/libfuse/libfuse

[32] GHC. 2022. GHC User's Guide. Retrieved October 2022 from https://downloads.haskell.org/ghc/latest/docs/users_guide/

[33] Havva Gulay Gurbuz and Bedir Tekinerdogan. 2018. Model-based testing for software safety: a systematic mapping study. *Software Quality Journal* 26, 4 (Dec 2018), 1327–1372. https://doi.org/10.1007/s11219-017-9386-2

[34] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. 2005. A principled approach to operating system construction in Haskell. In *Proceedings of the 10th International Conference on Functional Programming*. Tallinn, Estonia, 116–128.

[35] HaLVM. 2018. The Haskell Lightweight Virtual Machine (HaLVM) source archive. Retrieved October 2022 from https://github.com/GaloisInc/HaLVM

[36] Gernot Heiser, June Andronick, Kevin Elphinstone, Gerwin Klein, Ihor Kuz, and Leonid Ryzhyk. 2010. The Road to Trustworthy Systems. In *ACM Workshop on Scalable Trusted Computing (ACMSTC)*. ACM, Chicago, IL, USA, 3–10.

[37] Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *International Conference on Functional Programming*. 455–468.

[38] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *A List of Successes That Can Change the World*. Lecture Notes in Computer Science, Vol. 9600. Springer, 169–186.

[39] Steve Klabnik and Carol Nichols. 2017. *The Rust Programming Language*. No Starch Press.

[40] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70.

[41] 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 *ACM Symposium on Operating Systems Principles*. ACM, Big Sky, MT, USA, 207–220.

[42] Edward A. Kmett. 2022. lens: Lenses, Folds and Traversals. Retrieved August 2022 from https://hackage.haskell.org/package/lens

[43] Pieter Koopman, Peter Achten, and Rinus Plasmeijer. 2012. Model Based Testing with Logical Properties versus State Machines. In *Implementation and Application of Functional Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 116–133.

[44] Peter Lammich. 2013. Automatic Data Refinement. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*. Lecture Notes in Computer Science, Vol. 7998. Springer, 84–99.

[45] Peter Lammich and Andreas Lochbihler. 2018. Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches. *Journal of Automated Reasoning* (Mar 2018). https://doi.org/10.1007/s10817-018-9461-9

[46] 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 *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 114–129. http://doi.acm.org/10.1145/3009837.3009868

[47] Leonidas Lampropoulos and Benjamin C. Pierce. 2022. *QuickChick: Property-Based Testing in Coq*. Retrieved October 2022 from https://softwarefoundations.cis.upenn.edu/qc-current/index.html

[48] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 108–118. http://doi.acm.org/10.1145/325694.325708

[49] David R. MacIver. 2016. *Integrated vs Type-based Shrinking*. Article. Retrieved October 2022 from http://hypothesis.works/articles/integrated-shrinking

[50] David R. MacIver. 2016. QuickCheck in Every Language. Retrieved October 2022 from https://hypothesis.works/articles/quickcheck-in-every-language

[51] Toshiyuki Maeda. 2015. Kernel Mode Linux: Execute user processes in kernel mode. Retrieved October 2022 from http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/

[52] Carroll Morgan. 1990. *Programming from Specifications* (2nd ed.). Prentice Hall.

[53] Wojciech Mostowski, Thomas Arts, and John Hughes. 2017. Modelling of Autosar Libraries for Large Scale Testing. In *Workshop on Models for Formal Analysis of Real Systems (MARS@ETAPS)*. 184–199. https://doi.org/10.4204/EPTCS.244.7

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

[55] Liam O'Connor. 2019. *Type Systems for Systems Types*. Ph. D. Dissertation. UNSW, Sydney, Australia. http://handle.unsw.edu.au/1959.4/64238

[56] 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 *International Conference on Functional Programming*. Nara, Japan.

[57] Bryan O'Sullivan. 2022. aeson: Fast JSON parsing and encoding. Retrieved August 2022 from https://hackage.haskell.org/package/aeson

[58] Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming* 31 (2021).

[59] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. 2016. A Framework for the Automatic Formal Verification of Refinement from Cogent to C. In *International Conference on Interactive Theorem Proving*. Nancy, France.

[60] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. ACM, New York, NY, USA, 37–48. http://doi.acm.org/10.1145/1411286.1411292

[61] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *International Conference on Certified Programs and Proofs*. Los Angeles, CA, USA, 14–27.

[62] Jacob Stanley. 2022. *Hedgehog will eat all your bugs*. Open Source Project. Retrieved October 2022 from https://github.com/hedgehogqa/haskell-hedgehog

[63] Jan Tretmans. 2011. *Model-Based Testing and Some Steps towards Test-Based Modelling*. Springer Berlin Heidelberg, Berlin, Heidelberg, 297–326. https://doi.org/10.1007/978-3-642-21455-4_9

[64] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-Based Testing Approaches. *Softw. Test. Verif. Reliab.* 22, 5 (aug 2012), 297–312. https://doi.org/10.1002/stvr.456

[65] Dale Vaillancourt, Rex Page, and Matthias Felleisen. 2006. ACL2 in DrScheme. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '06)*. ACM, New York, NY, USA, 107–116. http://doi.acm.org/10.1145/1217975.1217999

[66] Philip Wadler. 1990. Linear types can change the world!. In *Programming Concepts and Methods*.