**Jonas Fiala**

# Property-based Testing for Hardware

Computer Science Tripos – Part II

Trinity Hall

10th May 2020

# Declaration

I, Jonas Fiala of Trinity Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Jonas Fiala of Trinity Hall, am content for my dissertation to be made available to the students and staff of the University.

Signed

Date

10th May 2020

# Proforma

| | |
|---|---|
| Candidate Number: | **2325B** |
| Project Title: | **Property-based Testing for Hardware** |
| Examination: | **Computer Science Tripos – Part II 2020** |
| Word Count: | **10590**[1] |
| Final line count: | **1754**[2] |
| Project Originators: | Matthew Naylor |
| Supervisor: | Matthew Naylor |

## Original Aims of the Project

The aim of the project was to create a library for testing hardware modules. It will require the user to specify some invariants about their design, these are exhaustively checked by the library to hold in all instantiations. Testing of modules can therefore be automated. This tool is to be created in an open-source hardware description language called Blarney. A suite of examples should also created to supplement documentation and to evaluate the usefulness of the library in uncovering bugs. A real world example may also be tested.

## Work Completed

The full library is implemented and a suite of nine examples has been created, all of which can successfully synthesize to an FPGA. Therefore, I have met all of the original requirements. Additionally, I extended the library to support testing of stateful modules. During evaluation I discovered undocumented failures of a module taken from a pre-existing project.

[1]This word count was computed using TeXcount (https://app.uio.no/ifi/texcount/)

[2]This line count was computed using `wc -l <relevant files>`

## Special Difficulties

None.

# Contents


**1 Introduction** **7**
1.1 Motivation . . . . . 7

**2 Preparation** **9**
2.1 Property Based Testing . . . . . 9
2.2 Hardware verification . . . . . 10
2.3 Requirements Analysis . . . . . 11
2.3.1 Haskell . . . . . 11
2.3.2 Blarney . . . . . 11
2.3.3 Quartus . . . . . 12
2.4 Success Criteria . . . . . 13
2.4.1 Starting Point . . . . . 13

**3 Implementation** **15**
3.1 Combinatorial Checking . . . . . 15
Mark 1: Fixed length bit vectors . . . . . 15
Mark 2: Universal quantification over lists . . . . . 17
Mark 3: Custom Generator . . . . . 18
Mark 4: Stateful TestBench . . . . . 20
3.2 Stateful property testing . . . . . 22
3.3 Extended library implementation . . . . . 24
3.3.1 State space . . . . . 24
3.3.2 Saved state . . . . . 26
3.3.3 Execution details . . . . . 26
3.4 Useful Additions . . . . . 28

**4 Evaluation** **31**
4.1 Sample properties . . . . . 31
4.2 Case study . . . . . 33
4.3 Experimental setup . . . . . 35

4.4 Simulation and Synthesis . . . . . 36
4.4.1 Comparing BlarneyCheck and BlueCheck . . . . . 40

**5 Conclusion** **41**
5.1 Further work . . . . . 41

**Bibliography** **43**

**A Project Proposal** **45**

# Chapter 1

# Introduction

Test benches have become widespread when developing hardware and so techniques for automatic testing and generation of test cases are critical. In my project I have created a library which exhaustively checks that a set of user-specified invariants always holds, reporting back a counterexample if that is not the case. This work is inspired by similar property-based testing tools which are now commonplace in software verification, but not yet in the open-source hardware community. I additionally extended my library to work with invariants on stateful (sequential) logic modules. This means that, in addition to simple logic, anything from a stack to a processor implementation can be tested, thus I met all of the core requirements and one important extension.

## 1.1 Motivation

Hardware verification tools are usually very heavyweight programs leading to verification being a complex process. Additionally these tools are often commercial products and expensive, so the open-source community needs new approaches.

Property-based testing has become popular in the software community. Both QuickCheck [1] and SmallCheck [2] are existing testing libraries that encourage lightweight formal specification of software through the reward of automatic testing and small counter-examples. However, such an approach has rarely been used for hardware.

This project aims to tackle some of the challenges with hardware verification and continue the work of BlueCheck [3] (a similar property-based testing library for the Bluespec HDL), but explore some new avenues. Namely, I use an open-source Bluespec-like HDL, called Blarney [4], developed here in the CL. Additionally, I use bounded exhaustive testing, which can can find minimal counterexamples without the need for

shrink steps. To my knowledge, no existing open-source property-based testing library for hardware supports bounded exhaustive testing.

# Chapter 2

# Preparation

First a background of property based and hardware testing is given. Then the tools used for the implementation of my project are presented and their suitability is analysed. The chapter ends with a brief description of the goals achieved and my experience prior to starting the work.

## 2.1 Property Based Testing

The idea of property based testing of software exploded in popularity in the early 2000s with the introduction of QuickCheck. Claessen et al. [1] present the idea of properties that are described as Haskell functions, which can be automatically tested. These properties are a kind of predicate logic with implicit universal quantification over all predicates. An example property from QuickCheck is given [1]:

```
prop_RevRev xs = reverse (reverse xs) == xs
```

This property asserts that twice applying the user defined function `reverse` should be the same as the identity. Making the programmer state such properties forces them to think about the fundamental properties of functions they define. Writing them explicitly is also a nice and compact form of documentation. However, the main benefit of properties with respect to testing is allowing it to be automated by a given library.

After a user has defined properties that are expected to always hold, QuickCheck will test these conjectures by randomly instantiating a small sample from the set of possible predicate values to test. It then reports back that either the properties held for all of the generated values or returns a falsifying counterexample. This can give the programmer assurance that in most cases their functions will behave correctly, but no guarantees can be made about completeness.

As the authors point out; a major limitation of QuickCheck is that there is no measurement of code coverage and only ad-hoc measurement of test case coverage [1]. Since then further development of tools like HPC [5] can give the user detailed code

coverage information. Even so, small failing test cases may be missed if other passing cases give full coverage. The QuickCheck library lets a user define custom generators to try and achieve proper test coverage, yet this may not be a simple task and still does not yield any guarantees.

Such potential shortcomings led to the development of SmallCheck. Runciman et al. [2] make the assumption that 'if a program does not fail in any simple case, it hardly ever fails in any case'. The library works on the same principle of properties which are asserted by the user — the difference is that, instead of randomly sampling values, all values are generated bounded by some depth. This bound is necessary since it is infeasible to test all possible values (some may be infinite). Thus, a depth must be defined for all values that can be generated and it should implicitly be correlated with the 'size' of a test case. This ensures that small counterexamples are found first and since a passing test is exhaustive to some depth it provides more confidence in the function under test.

Checking all test cases, even if bounded by some depth, allows SmallCheck to add some compelling new extensions to the property language. For example, existential quantifiers are supported; if testing ends and no witness has been found the user is notified that the assertion has failed and can proceed to search for satisfying cases at an increased depth. Such a property could never be expressed in QuickCheck. Note that with clever negation of universal quantification, existential quantifiers could already be expressed implicitly in SmallCheck. However, explicitly supporting existential quantification becomes particularly appealing when it is used in combination with universal quantification.

## 2.2 Hardware verification

Test benches are increasingly more common when developing hardware and so, many verification techniques have been developed. Generally, the most common method used is to generate a long formula which is then passed to a satisfiability modulo theory (SMT) solver for verification. Yet as Claessen [6] points out, calling an external theorem prover is a very heavyweight process. A variety of alternatives have been proposed.

Recent work [3] has successfully applied the idea of property based testing to hardware development in the Bluespec HDL. BlueCheck takes a similar approach to QuickCheck with random generation of test cases, though the nature of testing hardware leads to some fundamental differences between the two libraries. Primarily BlueCheck must differentiate between properties that mutate state and those that do not. Thus at least some minimal state must be saved within the test bench itself — for BlueCheck this is a state machine and a list of states visited since the last reset. If a failing test case is found then the whole list must be printed as any of the state transitions may have contributed to the final failure.

Although efficient at finding more 'obvious' bugs, errors which manifest in very rare cases are easily missed. Consequently an external theorem prover is still necessary to verify correctness. In the BlueCheck paper Naylor et al. [3] state that 'an exhaustive testing strategy ... such as that employed by SmallCheck, may be an attractive alternative to explore in future work'.

## 2.3 Requirements Analysis

### 2.3.1 Haskell

Haskell is a statically-typed lazy functional language. It is a very popular host language for embedded domain specific languages (DSL) as it supports higher order functions and has a flexible syntax. In the past 20 years many DSLs for hardware description have been developed for Haskell, notably one of the first was Lava [7], co-developed by one of the authors of QuickCheck.

Throughout the project I encountered a few problems trying to express my ideas correctly within the constraints of the type system, though this usually ended up revealing flaws in my design rather than shortcomings of the language. Thus learning Haskell on the fly did end up causing a few delays, but these were mostly brief and I was always able to find help.

### 2.3.2 Blarney

The entire testing library which I have created in this project is written for a Haskell DSL called Blarney. It is a modern variant of Lava [4, 7], supporting a more natural behavioural style of hardware description in addition to the structural style of Lava. In a behavioural style one can describe how data should flow from one clock cycle to the next and the synthesis tool must then figure out how to achieve this behaviour. Such register-transfer level (RTL) descriptions are much more natural and understandable in some cases. An example of this kind of description is given in Figure 2.1, describing a register which is assigned an incremented value on each clock cycle — there is no explicit structural wiring defined. In Blarney (`<==`) is used for register assignment, `val` returns the value of a register and (`.`) is defined as reverse function application rather than the usual function composition of Haskell.

```
top :: Module ()
top = do
  -- Create a register
  cycleCount :: Reg (Bit 4) <- makeReg 0

  always do
    -- Increment on every cycle
    cycleCount <== cycleCount.val + 1
    -- Display value on every cycle
    display "cycleCount = %0d" (cycleCount.val)
    -- Terminate simulation when count reaches 10
    when (cycleCount.val .==. 10) do
      display "Finished"
      finish
```

**Figure 2.1:** RTL style module, taken from Blarney examples [4]

Blarney is compiled, as any Haskell code may be, using the Glasgow Haskell Compiler (GHC). It uses a long list of Haskell extensions to allow for a more HDL-like syntax, but these are all specified by a script included with the code. Once compiled and executed, a Blarney module will output a folder containing the generated module in Verilog. If the code specifies, then a Makefile is also generated to simulate the module. This is done using a tool called Verilator, which converts the Verilog file into an executable ready for simulation. Alternatively, the Verilog file can be used in a Quarts project for synthesis. Communication over a UART is possible.

Using a little known and untested HDL for the entire project was a slight risk — there would be no help anywhere online. However, most features are well documented and my supervisor was able to help with any additional questions I had. Since Blarney is open-source I was also able to inspect the inner workings whenever necessary to gain a better understanding and when a feature that my library required was missing, it could easily be added. On the whole using Haskell and Blarney was a very rewarding experience.

### 2.3.3 Quartus

Quartus is Intel's proprietary software for creating hardware designs and programming Field Programmable Gate Arrays (FPGAs). In this project I used it to convert the Blarney generated Verilog into a synthesizable design and to program and communicate with an FPGA over a UART.

Although using Quartus is always a very heavyweight process, it was not needed for most of the project as simple simulation proved sufficient. Further, only a very limited subset of the full software was required for analysis of the synthesized test benches and comparison to simulation. Even so, setting up the project correctly in Quartus

proved to be somewhat of a challenge. After this there were no problems with using the compilation report for analysis and `quartus_dse` (design space explorer) for rigorous evaluation of area usage and maximum clocking frequency.

## 2.4 Success Criteria

With the project I set out to achieve the following:

- Implement a Blarney library for automatic testing of combinatorial circuits written in Blarney
- Devise a suite of example properties and buggy circuits, to evaluate the effectiveness of the library
- Measure the proportion of bugs reported and the size of counter-examples found
- Synthesise a test bench to an FPGA and compare speeds against a simulated test bench

All of these have been successfully completed with the additional extension of supporting stateful modules, allowing for exhaustive testing of all possible sequences of inputs, bounded by the maximum length of sequences tested.

### 2.4.1 Starting Point

Prior to working on the project I had no experience with programming in Haskell; the only functional programming language that I had used was ML from the Foundations of Computer Science course. For the hardware part of the project I had a little experience from the Digital Electronics and Computer Design courses. Therefore an important part of the project was learning to use the required tools.

My work extends the efforts of the Blarney library to provide a fully-featured open-source HDL. At the time of writing this project provides the only verification library for Blarney. In line with other similar testing libraries it is named BlarneyCheck.

# Chapter 3

# Implementation

In this chapter I present the library gradually, introducing key ideas one at a time accompanied by relevant examples. The library is built up as a number of versions from a simple core; however, this does not necessarily reflect the order in which features were actually implemented. In the final sections (3.2 onward) I demonstrate the main extension of the library along with a few small additions.

## 3.1 Combinatorial Checking

The problem of testing arbitrary properties describing combinatorial modules is split into three parts: the high level `check` function, the `TestBench` data structure to which properties are compiled, and test `Generator`s used for universal quantification. To increase the range of valid properties, custom user defined `Generator`s are possible and demonstrated.

### Mark 1: Fixed length bit vectors

At the heart of the testing library is the definition of the language of properties allowed. A `Prop` datatype is defined, allowing for a first order logic style of properties. A very simple example of such a definition of the datatype is given in Figure 3.1.

```
data Prop = Assert (Bit 1) | Forall (Bit 8 -> Prop)
```

**Figure 3.1:** Mark 1 property language

Valid properties in this language take the form of a combinatorial function on 8-bit wide variables, resulting in a 1-bit truth value, preceded with a list of `Forall` applications to bind all the free variables.

```
1  check :: Integer -> (String, Prop) -> Module ()
2  check maxVal (name, property) = always do
3    pass <- (checkGen 0 name property true)
4    when pass (display "Passed")
5    finish
6    where
7      checkGen _ fName (Assert val) disp = do
8        when (val.inv .&. disp) (display "Failed: " fName)
9        return (val .&. disp) -- Feed back 'pass' signal
10     checkGen currVal fName prop@(Forall f) disp = do
11       let appName = (fName ++ " " ++ show currVal)
12       passed <- checkGen 0 appName (f $ fromInteger currVal) disp
13       if (currVal >= maxVal) then return passed else
14         checkGen (currVal+1) fName prop passed
```

**Figure 3.2:** Mark 1 check function

As an example the commutativity property of summation is expressed as a `Prop` (keywords exported by BLARNEYCHECK are **underlined**):

```
let prop_Comm = Forall \x -> Forall \y -> Assert (x + y .==. y + x)
```

This is a direct translation from the first order logic expression that

$$\forall x.\forall y.x + y = y + x \tag{3.1}$$

and as the operators `+` and `.==.` are built in functions, this property can be expected to always hold. When the property is passed to the check function in Figure 3.2, we can see that this is indeed the case.

```
> check (2^7-1) ("Commutativity", prop_Comm)
Passed
```

This first version of the check function in 3.2 demonstrates how all test cases are generated in `checkGen` and a pass signal is returned. This pass signal is used to suppress further failing cases if a smaller one is found. A simple failing example can be demonstrated by the flawed assumption of commutativity of subtraction:

```
let prop_SubComm = Forall \x -> Forall \y -> Assert (x - y .==. y - x)
> check (2^7-1) ("SubCommutativity", prop_SubComm)
Failed: SubCommutativity 0 1
```

A single minimal failing example is printed. However even such a simple example highlights the flaw of generating all testing sequences to be checked in parallel; if the `maxVal` is set to be $(2^8-1)$ which translates to a mere $2^{16}$ test cases, the resulting verilog file is 35.7MB in size with 524289 individual wires declared and assigned. Clearly if even a slightly complicated module is to be tested it is infeasible to synthesize let alone

simulate all test cases in parallel. This is a major difference between exhaustive testing for software and hardware and is discussed in more detail in the Mark 4 version.

## Mark 2: Universal quantification over lists

Testing properties that are limited to universally quantifying over some small number of fixed width variables is quite unsatisfactory. The first extension is to allow universal quantification over lists. To this extent an additional `Prop` constructor is declared. The new definition of Prop is shown in Figure 3.3.

```
data Prop = Assert (Bit 1) | Forall (Bit 8 -> Prop)
          | ForallList Int ([Bit 8] -> Prop)
```

**Figure 3.3:** Mark 2 property language

This allows the property language to be more expressive, extending the range of testable modules. One example that can now easily be tested is a sorting network, which provides the function `sort :: [Bit n] -> [Bit n]`, where `n`, the bit width, is an arbitrary known natural number, unified to 8 by the property language. A helper `isSorted` function is defined and the property is expressed as in Figure 3.4.

```
isSorted :: [Bit 8] -> Bit 1
isSorted xs = andList $ zipWith (.<=.) xs (drop 1 xs)

let prop_isSN = ForallList 4 \xs -> Assert (isSorted (sort xs))
```

**Figure 3.4:** Example property with universal quantification over lists

However, the new `ForallList` must unfortunately diverge slightly from the idea of universal quantification; it also requires an `Int` to specify the length of lists to be tested. The exact list length of all test cases must be specified because at synthesis time the length of the list must be constant; a sorting network, by definition, only takes fixed size inputs.

It would be possible to instantiate the property for each possible length of list up to the maximum list length specified; however, this still requires a maximum list length to be given and one must ideally strive to minimise the number of instantiations. Also, testing only the maximum length list covers the test cases of shorter lists in most use cases, where the tests are run with all the additional values set to 0. As a result the trade-off was taken to sacrifice total coverage for much better FPGA area usage/simulation time. If tests with shorter list lengths are required then these can still be performed by setting the list length of the `ForallList` to the desired value.

## Mark 3: Custom Generator

When reading the previous section one may realise that a huge optimisation is missed. When dealing with sorting networks the zero-one principle [8] applies, which asserts that the network is valid if it is valid for lists of 1 bit wide vectors. Yet, due to the restrictions of the property language, lists of 8 bit wide vectors are generated and tested, drastically limiting the maximum list length that can realistically be chosen. So the generality of the universally quantified variable is extended. To this end BLARNEY provides the `Bits` class. This is the class of all types which can be represented in hardware, for example stored in a register or wire.

A new class called `Generator` is created and the definition is given in Figure 3.5. Any element of `Bits` can also have a `Generator` instance defined for it. This instance allows it to be *generated* for use in testing.

```
class Bits a => Generator a where
    initial :: a
    next    :: a -> a
    isFinal :: a -> (Bit 1)
```

**Figure 3.5:** Generator class

Two `Generator` instances have been pre-defined as part of my library for any element of the `Bits` class. The first is one that instantiates values in order of increasing size; meaning that this `Generator` has an `initial` value of 0, the `next` function is $+1$ and `isFinal` is true when all bits are one. Such an instantiation guarantees minimal counter-examples are found and is the simplest and most intuitive. In may cases this is a good choice and is thus provided as the default instance, i.e. if the user does not define their own Generator instance for a data type, then the default instance is used. Haskell's `OverlappingInstances` extension has been used to enable this behaviour; this built-in `Generator` instance is marked as `{-# OVERLAPPABLE #-}` meaning that user defined instances are allowed. In practice GHC always selects the most specific instance it can, so that a custom `Generator` is chosen if one is defined.

The second instance generates values in a random but systematic manner. It uses a Linear Congruential Generator (LCG) [9] to generate `next` values, with the LCG multiplier and increment constants chosen to ensure that all possible values of the given bit width are covered. The `initial` value can be any arbitrary value, since the final check is defined like this: `isFinal current = next current .==. initial`. This generator is the first step towards fully random test sequence generation in the likes of BLUECHECK. However, as this has already been done by such libraries, the area of random testing is not explored any further and this second `Generator` is simply left as an example of the possibilities of custom generators. Instead, exhaustive testing of stateful circuits is explored and covered in Chapter 3.2.

Both pre-defined `Generators` will exhaustively generate all possible values of the given bit width; however, in some cases this is undesirable or unnecessary. As an example the case of memory addresses is given; imagine that a designer wishes to test a memory management unit (MMU), and wants to generate lots of addresses that lie in different pages. In this case the page offset is redundant - generating multiple addresses to the same page would simply slow testing. A custom Generator is required. The designer has defined memory addresses to be of the type: `newtype MemAddr = MemAddr (Bit 32) deriving (Generic, Bits)`, for which a custom `Generator` is given in Figure 3.6.

```
instance Generator MemAddr where
    initial = unpack (constant 2^12-1)
    next current = unpack $ pack current + 2^12
    isFinal current = pack current .==. ones
```

**Figure 3.6:** Custom Generator for Memory Addresses[1]

A custom generator such as this enables testing with universal quantification over the entire range of 32-bit memory addresses. An example Property, showing that the lower 12 bits are constant, is given:

```
let prop_POConst = Forall \(MemAddr ma :: MemAddr)
                    -> Assert (slice @11 @0 ma .==. ones)
```

An exhaustive enumeration of all 32-bit memory addresses as a simple Bit vector would not be feasible in simulation otherwise. To allow universal quantification over any datatype with a `Generator` defined for it, the property language is extended once more as shown in Figure 3.7.

```
{-# LANGUAGE GADTs #-}
data Prop where
    Assert     :: Bit 1 -> Prop
    Forall     :: Generator a => (a -> Prop) -> Prop
    ForallList :: Generator a => Int -> ([a] -> Prop) -> Prop
```

**Figure 3.7:** Mark 3 property language

The Generalized Algebraic DataTypes (GADTs) Haskell extension is used to allow locally-quantified type variables. This new definition of `Prop` changes the language of properties very little as compared to the previous section, especially when using the built-in default generator. The only exception is that the user must sometimes define

[1]Since the `MemAddr` type derives from `Bits` one can use the **pack** and **unpack** keywords instead of using the `MemAddr` constructor

the bit width they wish to test as it is no longer fixed to 8. This is evident when rewriting the sorting network property from the previous section — it must now be defined as in Figure 3.8.

```
let prop_isSN = ForallList 26 \(xs :: [Bit 1]) ->
                                  Assert (isSorted (sort xs))
```

**Figure 3.8:** Sorting Network property, employing the zero-one principle to enable a drastic increase in the list length tested

In summary, the user can define custom generators for their data types in cases where they want more control, but if they don't, then the library's built-in default generator will be used, which works for any type in the `Bits` class.

## Mark 4: Stateful TestBench

The previous sections have all in some way been related to previous work done in property based testing. Exhaustive testing for hardware nevertheless raises a new issue. Generating a separate instance of the property in hardware for every possible test case is simply not feasible for most non-trivial testing scenarios; the module is too large to simulate or even synthesize. A new approach is required; test cases are generated sequentially, with one test per clock cycle.

```
data TestBench = TestBench {
    increment   :: Action ()
  , isDone      :: Bit 1
  , failed      :: Bit 1
  , displayFail :: Action ()
}
```

**Figure 3.9:** `TestBench` definition

To achieve this the `TestBench` datatype is defined using the Haskell record syntax as in Figure 3.9. This construct provides an interface for the `check` function to interact with individual properties in a sequential manner. During normal execution the `increment` action is performed every clock cycle until the `isDone` signal is high[2] at which point the test finishes with a successful result. Conversely, if ever the `failed` bit is high, testing is halted and the `displayFail` action is performed, outputting the failing state.

[2]If `increment` is executed when `isDone` is high then the state should be reset to the initial value, this is required in Section 3.2

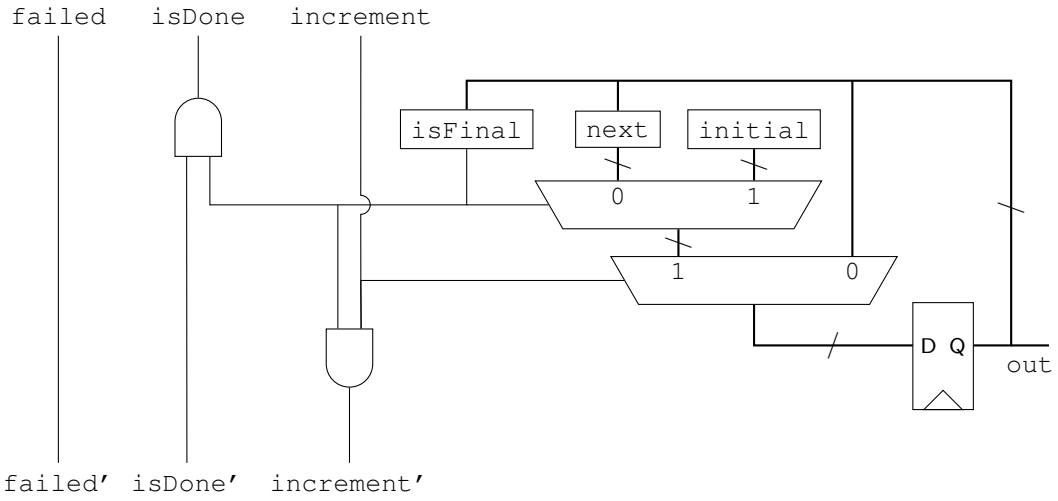


**Figure 3.10:** `TestBench` generated from `Forall`, the `failed` signal is simply passed up from `Assert`

Functions to map a `Prop` to a `TestBench` are provided by the library, and are the only requirement for adding a new constructor to the `Prop` datatype. This mapping is performed recursively on the structure of `Prop` where each level creates its own `TestBench` also incorporating the one from below if necessary.

The lowest level, which terminates the sequence of constructors, will always be an `Assert` statement with a binary value. This is trivially mapped to the four records of a `TestBench`, with the only one of note being the `failed` signal which is simply the inverse of the asserted binary value. The other constructors (`Forall` and `ForallList`) build appropriately on top of this base case. As an example, the complete `TestBench` logic generated for a `Forall` statement is given in Figure 3.10. In this figure, the records at the bottom, marked prime, lead to/from the test bench below, and the ones at the top are passed above. The `out` signal is wired to the property using function application. Additionally, the use of all three generator functions is depicted.

The case of `ForallList` is surprisingly straightforward. It is simply mapped to repeated `Forall` test benches stacked on top of each other and the `out` wires from all are concatenated to make a list. The number of repetitions is given by the maximum list length which is a part of `ForallList`.

There remains one final detail; a user will realistically wish to check multiple `Prop`s at once and get feedback on which one, if any, failed. To accommodate this, a new helper type is defined:

```
type Property = (String, Prop)
```

and the check function has the type

```
check :: [Property] -> Module (Bit 1)
```

Note that the Integer `maxVal` required in the Mark 1 check function is no longer needed, since `Generators` now provide the necessary functionality. The binary output is high once checking completes.

From the properties a list of test benches is created using the mapping described above. All of these test benches are logically combined into one: `failed` signals ORed together, `isDone` signals ANDed and all `increment` actions executed together in parallel. The resulting single interface is then easily tested within the `check` function.

## 3.2 Stateful property testing

In BLARNEY the `Action` monad specifies what to do within one clock cycle. Alternatively, a `Recipe` can specify work over multiple clock cycles. The extension to the `Prop` type in Figure 3.11 provides the additional necessary constructors to allow the library to test stateful properties. It is important to note that these can still be combined with the `Forall` and `ForallList` constructors.

```
data Prop where
  Assert     :: Bit 1 -> Prop
  Forall     :: Generator a => (a -> Prop) -> Prop
  ForallList :: Generator a => Int -> ([a] -> Prop) -> Prop
  WhenAction :: Bit 1 -> Action () -> Prop
  WhenRecipe :: Bit 1 -> Recipe -> Prop
```

**Figure 3.11:** Stateful property language

The binary value of both new `Props` acts as a guard bit for when the action should or should not be performed, such as guarding against popping from an empty stack.

As an example consider that the user has just created a stack implementation which caches only the top element in a register and uses block RAM to store other values. Previously they had been using an implementation which stored all values in registers, this was easy to implement and is more obviously correct than the new version, but only supported smaller stack sizes. Both implement a standard interface given in Figure 3.12 and are created with a maximum size of $2^5$.

```
data Stack a = Stack {
    push    :: a -> Action ()
  , pop     :: Action ()
  , top     :: a
  , isEmpty :: Bit 1
  , clear   :: Action ()
}
stkGolden :: Stack (Bit 5) <- makeGoldenStack 5
stkBRAM   :: Stack (Bit 5) <- makeBRAMStack 5
```

**Figure 3.12:** Stack interface and creation

The original implementation is taken as the golden model and assumed to always be correct, thus to prove the correctness of the BRAM stack one must simply assert that it 'behaves' the same way as the golden model in all situations. That is, the `isEmpty` signal of both is equal and when `isEmpty` is false then `top` is equal. Such assertions are easily stated in Figure 3.13 using already known property constructors.

```
let topEq = stkGolden.top .==. stkBRAM.top
let prop_TopEq = Assert (stkGolden.isEmpty .|. topEq)
let prop_EmptyEq = Assert (stkGolden.isEmpty .==. stkBRAM.isEmpty)
```

**Figure 3.13:** Stack invariants

These two assertions must hold regardless of the sequence of `push` and `pop` actions taken. For stateful testing a reset action is needed to clear the effects of other actions and return to the initial state. In this case `clear` is used and so will not be explicitly tested. Implicitly if the reset is incorrectly implemented then testing will usually fail anyway, but in such cases the report of the failure may not provide the full picture.

To test this, two extra properties are defined in Figure 3.14 using the new `WhenAction` constructor[3]. This tells the `check` function which actions it should be able to take without breaking the invariants.

```
let prop_Push = Forall \x -> WhenAction true
                                  (push stkGolden x >> push stkBRAM x)
-- No need to guard (stkBRAM.isEmpty.inv) since equality is asserted
let prop_Pop = WhenAction (stkGolden.isEmpty.inv)
                                  (pop stkGolden >> pop stkBRAM)
```

**Figure 3.14:** Stack actions

[3]The `(>>)` monadic composition operator is used to perform multiple actions in the same clock cycle

Finally, the full test bench is created by calling the `check` function which has been updated to take a reset action and the maximum sequence length to test, along with the list of properties[4].

```
let stackProperties = [
  ("TopEq", prop_TopEq),
  ("EmptyEq", prop_EmptyEq),
  ("Push", prop_Push),
  ("Pop", prop_Pop)
]
let reset = stkGolden.clear >> stkBRAM.clear

> check stackProperties reset 7
- All tests passed to depth 0 at time 0 -
- All tests passed to depth 1 at time 66 -
- All tests passed to depth 2 at time 3333 -
- All tests passed to depth 3 at time 147081 -
- All tests passed to depth 4 at time 6076686 -
=== Found failing case at depth 5 after 240673434 ticks: TopEq ===
0: Push 0x1
1: Push 0x0
2: Push 0x0
3: Pop
4: Pop
5: 'TopEq' fails
```

**Figure 3.15:** Declaration of arguments and Stack check execution

Now the user can see that there is actually a bug in the BRAM stack implementation: the sequence of actions taken is listed and the failing assertion is given. As it turns out this is the minimal counter example; all five actions in this order are required to expose the bug. In fact, as the output shows all test sequences of length 4 or less passed, thus whenever a failing sequence is found by BlarneyCheck, it is guaranteed to be minimal. To show this, an explanation of the test bench itself is necessary.

## 3.3 Extended library implementation

### 3.3.1 State space

It is useful to distinguish properties that mutate state (*impure* properties) from those that don't (*pure* properties) [3]. Thus the `TestBench` class is separated into two, a

[4]The Mark 4 version for combinatorial only logic is renamed to `checkPure`. The new function type is: `check :: [Property] -> Action () -> Int -> Module (Bit 1)`

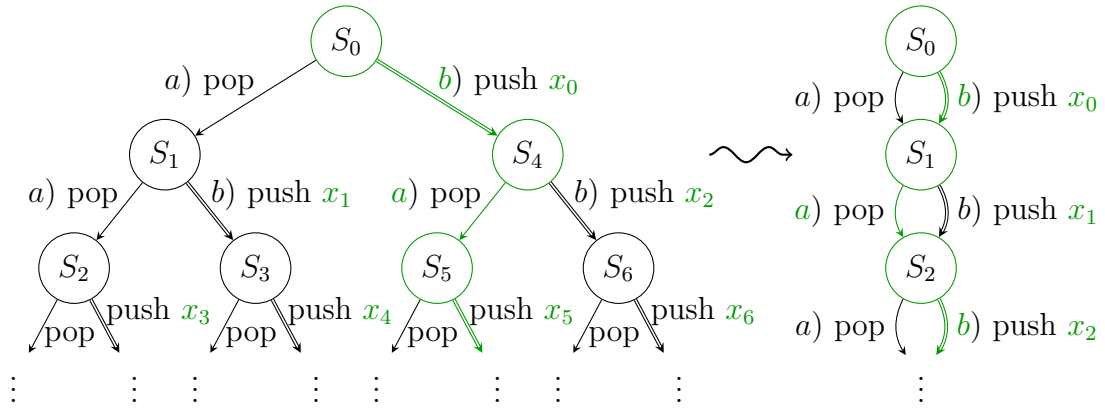


**Figure 3.16:** State space of stateful tester, highlighted edges are the ones currently taken

`PureTestBench`, equivalent to the old simple class, and a new `ImpureTestBench`. *Pure* properties which end in an `Assert` are mapped to `PureTestBench`es whereas *impure* ones ending in a `WhenAction` or `WhenRecipe` are mapped to `ImpureTestBench`es.

Consider each impure property as an edge from one state to another unique state. Each state represents an individual node in such a defined graph. Checking starts in the initial state $S_0$ and executing the given reset action will take the state back into $S_0$.

As there is no analysis of what individual impure properties 'do', each possible state must be treated as unique. Thus the state space is a tree with $S_0$ at the root and a branching factor of $b$, which is dependant on the number of impure properties given (for the stack example $b = 2$). To perform exhaustive checking, the entire state space is explored using Iterative Deepening (ID) up to the user-defined maximum depth. The depth is defined as the the number of edges taken from the root $S_0$. The tree of possible states for the stack example is shown in the left half of Figure 3.16.

Whenever a new node is visited, all of the pure properties are checked. That is, a full run of the stateful tester (Mark 4) is executed. For the current example both of the pure properties are simple `Assert` statements (`prop_TopEq` and `prop_EmptyEq` from Figure 3.13) so checking them will only take one clock cycle. Though it is not hard to imagine that in a more complex example visiting each new node may take much longer.

Since universal quantification is also allowed for impure properties (such as the `prop_Push` in Figure 3.14) the current argument at any edge must also be saved. This would be very costly in terms of registers for the entire state space, however it is possible to use only one register per impure property per level. This is illustrated in Figure 3.16; when exploring the tree naively $x_0, \cdots, x_6, \cdots$ would have to be saved, compared to only $x_0, \cdots, x_2, \cdots$ with the flat tree — an exponential reduction.

Each edge (impure property) is mapped to a corresponding `ImpureTestBench`. This mapping will not be described in detail but is similar to the recursive map-

ping for `PureTestBenches`, where one test bench is stacked on top of another. The `ImpureTestBench` interface provides actions to traverse the given edge (say executing `push` $\texttt{x}_\texttt{n}$), to increment the current argument of the edge ($\texttt{x}_\texttt{n}$ to $\texttt{x}_\texttt{n}\texttt{.next}$) and to display the action taken upon failure. Binary values for when the current argument is the final one and when a `WhenRecipe` is finished executing are also provided as part of the interface.

### 3.3.2 Saved state

It is evident that the minimal state required is the choice of edge taken at each node and the arguments passed to `Forall` properties. To correspond with the flat tree in Figure 3.16, the name

$$S_i^{\text{edge}} \in \{a, b, \ldots\} \tag{3.2}$$

is given to the current choice of edge from node $S_i$ (in the figure $a$ corresponds to pop and $b$ to push, so $S_0^{\text{edge}} = b$, $S_1^{\text{edge}} = a$, $S_2^{\text{edge}} = b$,...), with $i$ ranging from 0 to $n$, the current maximum depth. This value is derived from the ID algorithm, starting at 0 and increasing up to the user-specified maximum sequence length.

The notation $S_i^x$ is used for the edge (`ImpureTestBench`) itself, specified by $x \in \{a, b, \ldots\}$. And

$$S_i^x.\text{args} \in \{[\,], [\texttt{initial}], [\texttt{next initial}], \ldots, [\texttt{initial}, \texttt{initial}], \ldots\} \tag{3.3}$$

gives the list of arguments currently applied to the given impure property. The length of the list corresponds to the number of `Forall` constructors used[5]. So `prop_Pop` from Figure 3.14 has none thus $\forall i. S_i^a.\text{args} = [\,]$, compared to `prop_Push` with one meaning that $S_i^b.\text{args} = [x_i]$. It is important to note that for example $S_1^b.\text{args} = [x_1]$ will still require a register to store $x_1$ even though currently $S_1^{\text{edge}} = a$ (all $x_0$ to $x_n$ must be stored).

To save the state of both $S_i^{\text{edge}}$ and $S_i^x.\text{args}$ for all $x$ and $i$ a circular buffer is employed, meaning $i$ is implicitly stored by the current position within the buffer. A buffer of size $n$ is used to ensure that $i$ rolls around to 0 when $n$ is reached. The built-in BLARNEY `Queue` is used as the underlying implementation of the circular buffer. Whenever an edge is taken, the buffer is *cycled* by issuing an `enq` simultaneously with a `deq`. The whole structure is shown in Figure 3.17.

### 3.3.3 Execution details

At first $n$ is set to 0 and execution proceeds exactly the same as with the Mark 4 version; all `Assert` properties are exhaustively checked. Each time that all test cases are exhausted and no failing case has been found $n$ is increased and an additional value

[5] Any `ForallList` constructors are simply flattened into multiple `Foralls`, similarly to the approach taken for pure properties described in Mark 4.

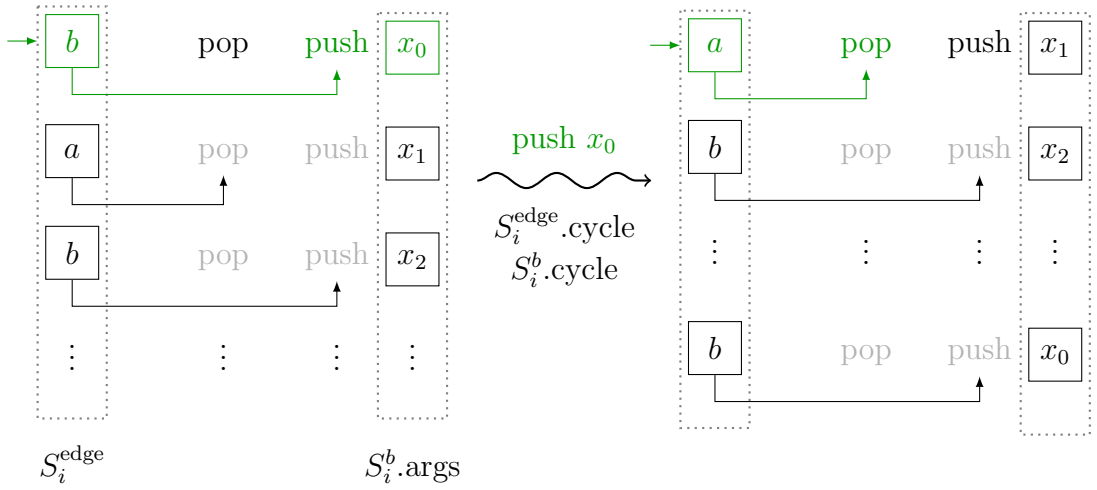


**Figure 3.17:** Circular buffer usage

is enqueued onto all queues (to ensure they remain of size $n$). Once $n$ reaches the maximum value the test concludes with success.

Since the state space is searched using ID, when sequences of length $n$ are being checked, all sequences of length $n-1$ and below must have already passed. This means it is only necessary to check the pure properties once a sequence of exactly $n$ impure actions has been taken. After which the `reset` action takes the state back to $S_0$, in preparation for the next sequence of $n$ impure actions.

To minimise the clock cycles spent on each sequence, the arguments are incremented whilst executing the sequence of impure actions. Such an increment is achieved by enqueueing the next value rather than the current one at the front of the queue. This is done instead of the regular cycle. However, great care must be taken when incrementing to the next sequence; a failing case will only be identified after the whole sequence has been executed, thus if the next sequence is already enqueued then the failing one cannot be displayed (remember that `Generators` only define a `next` function). To work around this issue, the value at the front of the queue is always incremented first, this new value is used for executing the impure action and then enqueued back.

For convenience call the edge which is currently selected $S_i^{\mathbf{S}} = S_i^{(S_i^{\text{edge}})}$. The execution of one test sequence is as follows

1. Start at state $S_0$ using the reset action if required. All queues have element $i = 0$ on top.
2. Increment $S_i^{\mathbf{S}}$.args, and cycle all other queues.
3. If the value of $S_i^{\mathbf{S}}$.args *before* incrementing was final then increment $S_i^{\text{edge}}$, else just cycle.

4. Execute $S_i^{\mathbf{s}}$, this is with either incremented args, or incremented **s**. Exactly one will be true.

5. If $S_i^{\text{edge}}$ *before* incrementing was final then go to step 2 (keep incrementing) up to $n$ times.

6. If all edges were incremented and final (repeated step 5 $n$ times) then all test sequences of length $n$ have been exhausted, increment $n$ as described. Else keep cycling all queues without incrementing, each time executing $S_i^{\mathbf{s}}$ until $i = n$.

7. Exhaustively check all pure properties.

8. Repeat from step 1 until either a pure property fails, or $n$ reaches the maximum value.

Executing a `WhenRecipe` rather than a `WhenAction` may take multiple clock cycles, thus waiting until it is done before executing the next edge is necessary. However, both are treated as only one edge. If a failing assertion is found at step 7 then all the steps taken and arguments passed to get to that point are printed. This can be done by simply executing without any increments (starting from step 6) and displaying each edge executed.

## 3.4 Useful Additions

The current property language differs somewhat from that of BlueCheck. An important difference is that the basic assertion in BlueCheck is an equivalence between two values, compared to the one bit `Assert` of BlarneyCheck. Thus, when writing an equivalence assertion one must explicitly include the (`.==.`). For this reason a printed failing case from BlueCheck can include the two values which did not meet equality, whereas an `Assert` only sees a binary value. However, assertions over arbitrary binary values are naturally more flexible. Take the example from Figure 3.13: `Assert (stkGolden.isEmpty .|. topEq)`, it is possible to be lenient with this assertion when both of the stacks are empty; in such a case `top` is undefined and it can reasonably be anything.

Yet, getting more information about the failing case than just the fact that it failed can be very helpful to get to the root of the problem. To this end I add a minor extension to the property language: `Assert' :: Bit 1 -> Action () -> Prop`. This new constructor behaves identically to an `Assert`, but allows specifying an action to be performed upon failure. Most commonly this will be used to display important values. For example the assertion from Figure 3.13 could be extended with the following action: `(display stkGolden.top " v " stkBRAM.top)`.

Sometimes, this is still not enough to fully understand why something failed. Therefore BlarneyCheck also has a built in debug mode. It can be activated by running

the executable file generated using verilator with the `+DEBUG` plusarg, which is only possible in simulation. When executed this way, a lot of additional output is printed. Each output line will start with the sequence of impure edges taken and then display all of the pure properties verified. All sequences including non-failing ones will be displayed this way. This mode is useful to gain an understanding of what takes place under the hood and can be used to check that all of the invariants which one wished to assert are actually being verified.

Such a mode could even be used to verify that BlarneyCheck itself is working correctly. However, going through each tested sequence one by one would be far too slow. This is one of the reasons which motivated the development of a static analysis tool. The tool is included as the `estimateTestCaseCount` function and simply replaces the `check` function. This means that it can be used with the already defined list of properties passed to `check`. For example the execution of Figure 3.15 could instead be analysed as shown in Figure 3.18. This tool requires that all `Generator`s in use are extended with a `range` field, which specifies the number of values from `initial` to `isFinal`. For example, the default generator will return $2^n$, where $n$ is the bit width. Using this the tool can exactly calculate the number of clock cycles required to exhaustively check to any depth. Runtime values are only estimates and will vary based on the complexity of the module under test.

```
-- (check stackProperties reset 7) is replaced with:
> estimateTestCaseCount stackProperties 7
- If sim runs at 2MHz (20.9 bits) and native at 50MHz (25.6 bits) -
... small depths elided ...
Clock cycles to test to depth 4: 6076687 or 22.5 bits
Simulation would run for about: 3.038s
Synthesized testing would take: 0.121s

Clock cycles to test to depth 5: 240889045 or 27.8 bits
Simulation would run for about: 2m 0.444s
Synthesized testing would take: 4.817s

Clock cycles to test to depth 6: 9281164828 or 33.1 bits
Simulation would run for about: 1h 17m 20.582s
Synthesized testing would take: 3m 5.623s

Clock cycles to test to depth 7: 350228708644 or 38.3 bits
Simulation would run for about: 48h 38m 34.354s
Synthesized testing would take: 1h 56m 44.574s
```

**Figure 3.18:** Static analysis result

As the clock cycle count scales exponentially with the bit width of the variables

`Forall` quantifies over, it is practical to consider these values in the $\log_2$ domain. Call this value the *bit requirement* of a property (or list of properties), giving the number of bits which must be explored for an exhaustive check. Calculating this for a pure property is easy; it is simply the sum of the bit width from each `Forall`. For example the bit requirement of the `prop_isSN` from Figure 3.8 is 26, since the `ForallList` is unrolled into 26 `Forall`s over 1 bit values. For a list of pure properties the requirement is simply the maximum of any one of them, since all pure properties are checked in parallel. Call the bit requirement of all pure properties $b$.

It is slightly harder to compute for impure properties since the depth tested to $d$ is also a factor. Using the static analysis tool will give an exact result, however the bit requirement of a list of pure and impure properties can be estimated with the following equation

$$\text{bit requirement} \approx b' * d + \log_2(d+1) + b \tag{3.4}$$

where $b'$ is the bit requirement of impure properties, calculated as the softmax over individual impure properties, though it can be quite closely approximated by taking the maximum (just as $b$ is calculated for pure properties). Note that when depth $d = 0$, only pure properties are tested thus the bit requirement is $b$.

Once such a bit requirement is calculated, it proves invaluable for the analysis of 'resources' required to perform a full test. For example if one assumes that the stack test bench can be simulated at 2.2MHz, this gives a base rate of $\log_2(2.2e^6) \approx 21$ bits for one second of simulation. However, from Figure 3.18 we can see that, to exhaustively test to depth 5, 27.8 bits are required. Therefore another 6.8 bits must be achieved — one way to do this is to simply let the simulation run $2^{6.8}$ times longer (111s). Alternatively, the test bench could be synthesized and run natively on an FPGA at 160MHz thus achieving a base of 27.2 bits, an advantage of 6.2 bits over simulation. As is discussed in the next chapter, we can see that there are other ways to achieve even higher bit counts, without waiting an unreasonable amount of time.

# Chapter 4

# Evaluation

This chapter starts by introducing additional examples to complement the ones given in the previous chapter. Each of these sample modules highlights some aspect of testing enabled by the property language. A description of how measurements of speed, FPGA area and block RAM usage were obtained is given, and these results are analysed in depth. Lastly, a comparison is drawn to previous work.

## 4.1 Sample properties

A variety of unique testable modules were created throughout the course of the project, with each carefully chosen to provide an important point for analysis of the library. All of these examples are included along with the main library to supplement documentation and to provide a simple starting point for most other use cases. The full suite is as follows:

- *Sums* — The two properties of addition which must always hold are Associativity and Commutativity. An example of Commutativity expressed in the property language was given in the previous chapter. As this is testing a built in (+) function of Blarney, it is a basic requirement that the check always passes. These are useful properties for quantifying different metrics when comparing synthesized to simulated versions.

- *Parallel Sums* — A simple illustration of how custom `Generator`s can be used for ad-hoc parallel testing. The properties make an identical assertion to that from the *Sums* example, but take half the clock cycles to test.

- *First Hot Function* — This is an example taken from BlueCheck; the function under test returns a bit-string in which only the least significant non-zero bit of the input bit-string is set. The correctness and completeness of such a function can be defined using three properties:

- Exactly one output bit is set if input is nonzero, else no bits set (One Is Hot)
- No less-significant hot bits in input than that set in output (Hot Bit First)
- Hot bit in output is also hot in input (Hot Bit Common)

- *Sorting network* — A common hardware module is the sorting network. The property required to verify that one is correct has been discussed in the previous chapter; an `isSorted` function is defined, allowing an `isSortingNetwork` property to be asserted. Examples of both a correct bubble sorter and a buggy bitonic sorter are tested.

- Random generator — Properties to demonstrate the use of the in-built random generator and a definition of a user defined one are provided to show how this `Generator` can be used. All properties which can be tested using this could also be tested with the default `Generator`, providing much the same results, thus this example contains only a trivial assertion and is not discussed further.

- Memory addresses — Another example of defining a custom `Generator` is given in this example. This is the case where a designer wishes to test an MMU, by using lots of addresses which lie in different pages. Thus generating multiple page offsets in the same page is redundant and a custom generator leads to significantly faster testing. This example is given to illustrate the usefulness of custom `Generators` - the given property itself is trivial. Therefore no analysis of speed or area for this example is performed.

- *Simple Stack* — For this universal stateful component correctness in all cases is extremely important. A user-defined efficient version of the stack using BRAM is compared to a library-defined but inefficient implementation using registers only. If the new version behaves the same in all cases, this will guarantee correctness. This is the example used for the description of the stateful checking part of the library and contains a bug which can only be discovered after a minimum sequence of 5 actions.

- *Complex Stack* — The Actora [10] project centers around a stack based CPU, therefore it requires a rather complex stack implementation to function. The stack module used for the Actora core is taken directly and compared to another golden model, which is far more inefficient but provides the same interface with the use of registers only. Testing of such a real-world example revealed rather interesting behaviour, which could not be uncovered by testing the full core.

- *Toy CPU* — This example shows that even a complete, albeit simple, CPU can be rigorously tested. The module under test is the 8-bit 4 instruction CPU from Blarney, which is slightly adapted to allow for testing. The design claims data and control hazards are resolved, a statement which can be confidently proven after verification with BlarneyCheck.

In all the examples above, the reported cycles taken to perform exhaustive testing are exactly equal to the count predicted by the static analysis tool. This provides significant confidence in the correctness of the testing library itself.

## 4.2 Case study

Actora [10] was chosen to demonstrate the application of this library to a pre-existing complex project. Its main design centers around the Actora CPU — a processor, designed to run Erlang-like programs, implemented as a stack machine. No effort is made to dynamically identify instruction level parallelism (ILP), resulting in a small core which can be duplicated many times. This hardware parallelism is easily exploited by applications conforming to the actor model [11], where parallelism is explicit and without shared state [10].

The processor itself, being a stack machine, has strong requirements for the stack implementation. The current Actora stack supports up to two `push` operations or a `push` and `pop n` in parallel during one cycle. It also has a `copy n` operation which can bring an element from an arbitrary position in the stack to the top (indexed by $n$ from the top) or when called with a negative offset $n$, it retrieves elements previously popped. This way a slide operation can be achieved; first `pop m` and then $n$ times `copy (n-m-1)` to get $n$ items back (where $m > n$), effectively deleting elements $n$ to $m$ from the top. The interface[1] along with accompanying documentation is given in Figure 4.1.

```
-- Stack of 2^n items of type a
data Stack n a = Stack {
  -- Can push one or two items per cycle (push1 happens after push2)
    push1 :: a -> Action ()
  , push2 :: a -> Action ()
  -- Push nth item from top (n can be negative to implement a slide)
  , copy :: Bit n -> Action ()
  -- Pop any number of items (can be called in parallel with push1)
  , pop :: Bit n -> Action ()
  -- Size of the stack
  , size :: Bit n
  -- Top two stack values
  , top1 :: a
  , top2 :: a
}
```

**Figure 4.1:** Actora stack interface

[1]The real interface also provides underflow and overflow flags, however these require a different set of properties to test and are left out in this example.

For testing a golden model is defined, like with the simple stack example. This model is easy to define when using a list of registers and is much more obviously correct. For the `size`, `top1` and `top2` fields, *pure* equality properties are asserted. Then *impure* properties for all possible permutations of actions allowed in one clock cycle are created. To reset the state `(pop stack.size)` is used.

The first test run reveals `push2` cannot be used without also doing `push1` at the same time, otherwise no value is pushed. The output printed by BlarneyCheck is given in Figure 4.2. This, although not explicitly stated in the interface definition, is understandable; `push2` is only for the special case when two elements are to be pushed.

```
=== Found failing case at depth 1 after 12 ticks: Top2Eq ===
0: Push2 0x1
1: 'Top2Eq {1 v 0}' fails
```

**Figure 4.2:** Initial failing sequence

So the failing property is removed from testing, `push2` will now only be used in conjunction with `push1`. However, when the test bench is simulated again, another failing corner case is uncovered. This time when calling `pop 0` the wrong element appears at the top of the stack. Again this is not a 'real' bug; the stack designer can assume this case will never come up. Yet by using the testing library more undocumented and unexpected behaviour is uncovered. This strange behaviour happens since popping triggers a load from BRAM, but the current top value hasn't been written back yet and so is overwritten by a stale value. As one can see in Figure 4.3, the failing sequence is not entirely clear — the 3 at the top of the Actora stack is from a previous test sequence (which can clearly be seen by running the test bench in `+DEBUG` mode). Although modules should in general be made easily resettable, such a design is not always possible as is the case with BRAM. This shortcoming is pointed out by the authors of BlueCheck, but, due to the nature of exhaustive testing, resetting the design under test is absolutely crucial for BlarneyCheck.

```
=== Found failing case at depth 2 after 5851 ticks: Top1Eq ===
0: Push1 0x0
1: Pop 0x0
2: 'Top1Eq {0 v 3}' fails
```

**Figure 4.3:** Longer failing sequence

Surprisingly after `pop 0` is disallowed via the guard bit, one more failing sequence is revealed by the test bench. This time after attempting to `copy (-2)`. Which, unlike the previous examples, is a perfectly reasonable use case. The documentation states $n$

can be negative to retrieve elements previously on the stack — this is exactly what is attempted and fails. However, just as with the previous failing case, the printed failing sequence shown in Figure 4.4 is somewhat unclear. In fact the 3 must again be from a previous test sequence and implicates BRAM as the primary cause for the failure.

```
=== Found failing case at depth 3 after 484588 ticks: Top1Eq ===
0: Push1&2 0x0 0x0
1: Pop 0x2
2: Copy 0x6     -- The max stack size is 8, so this equates to (-2)
3: 'Top1Eq {0 v 3}' fails
```

**Figure 4.4:** Final failing sequence

After discussing the problem with the creator of the Actora core, I found out that this shortcoming can be explained due to the fact that a negative `copy` relies on the value having been written to BRAM but the top two elements are cached in registers and may not be written back. Thus a `copy` will retrieve arbitrary data, which happened to be there previously. The design of the core itself is actually aware of the limitation and will save the top two stack elements in registers before doing a `pop`. Nevertheless, finding all of these undocumented constraints using BlarneyCheck without any knowledge about the design under test was very promising.

Interestingly, a full reset of the BRAM between test sequences would mean that both of the previous failures would have taken a much longer sequence to find. This sequence however would be much more informative. After taking into account all three limitations found, testing shows that any sequence up to length 4 satisfies the constraints.

## 4.3 Experimental setup

This section describes how quantative measurements of runtime and FPGA resource usage were obtained. First, all of the example properties were compiled with `ghc`. The executed file calls the Blarney function `writeVerilogTop` which outputs a verilog file. This verilog module is then either simulated or synthesized. For simulation I use the `verilator` library and then execute the resulting binary file. Synthesis is done using `qsys-generate` after which I run `quartus_dse` with 15 seeds, this ensures a good coverage and gives a more accurate estimate of the maximum clock rate. The synthesis target is a Cyclone V FPGA used in the lab with a base clock frequency of 50 MHz. Both the simulated and synthesized versions are discussed in this chapter.

To obtain runtimes the simulated version is executed with the `time` command to get an approximation of the execution duration. As such a timing may be inaccurate, I repeated each measurement 16 times, and from this calculate the mean with appropriate

confidence intervals. For the rest of this chapter I will always use the mean value and plots also include the standard deviation. Any exact values given are appropriately rounded.

The simulation also includes an internal clock measuring the number of ticks taken to finish the testing, this value is reported on both success or failure. The measurement is used to calculate the clock frequency of simulation by dividing the number of clock ticks simulated by the runtime. The clock tick measurement is also used for analysis of the synthesised version.

Once synthesis is complete the timing results and area usage are analyzed with `quartus_dse --report exploration_summary`, in all cases the reported fmax (maximum safe clock frequency) had a coefficient of variation of under 10%. I then divide the the clock cycles taken to run the test bench (measured in simulation) by the maximum clock rate to calculate the synthesized runtime. Like with simulation, the mean and confidence intervals are calculated. The synthesized version of the project is further opened in Quartus and analysed using the Compilation Report window. This yields useful information about the usage of other important resources such a BRAM.

## 4.4 Simulation and Synthesis

This section uses the *bit requirement* value of properties discussed at the end of the previous chapter. I encourage readers to have read and understood that section first.

With the use of `verilator`, simulation achieves identical results to the FPGA sythesized version. It also enables easy implementation of a few nice to have features such as exact failing sequence output or the debug output mode. These are possible to implement on FPGA over UART, but would require a lot more work and are out of the scope of this project.

Simulation is also far more practical for lightweight use. Compilation takes only a few seconds before an executable file is ready, compared to the 3–6 minutes it takes to synthesize a design. Let alone all of the complexities of managing a Quartus project. Thus, for small properties with a bit requirement of under 30 bits simulation is sufficient.

On my computer simulating Associativity for three 10 bit inputs and Commutativity for two 15 bit inputs (bit requirement for both is 30, thus maximum is still 30) took 78s, at a simulated clock frequency of 13.8 MHz meaning 23.72 base bits. According to Quartus the synthesized version can clock up to 149 MHz or 27.15 bits, which means 2.85 bits of execution time (7.2s) to reach the 30 bit requirement. Thus running the test bench natively yields an advantage of 3.43 bits — a small difference compared to the other examples tested. This is caused by a relatively low fmax for such a simple module, the advantage is significantly greater for complex modules. Note that simulation makes up for these 3.43 bits by running $2^{3.43} \approx 10.8$ times longer.

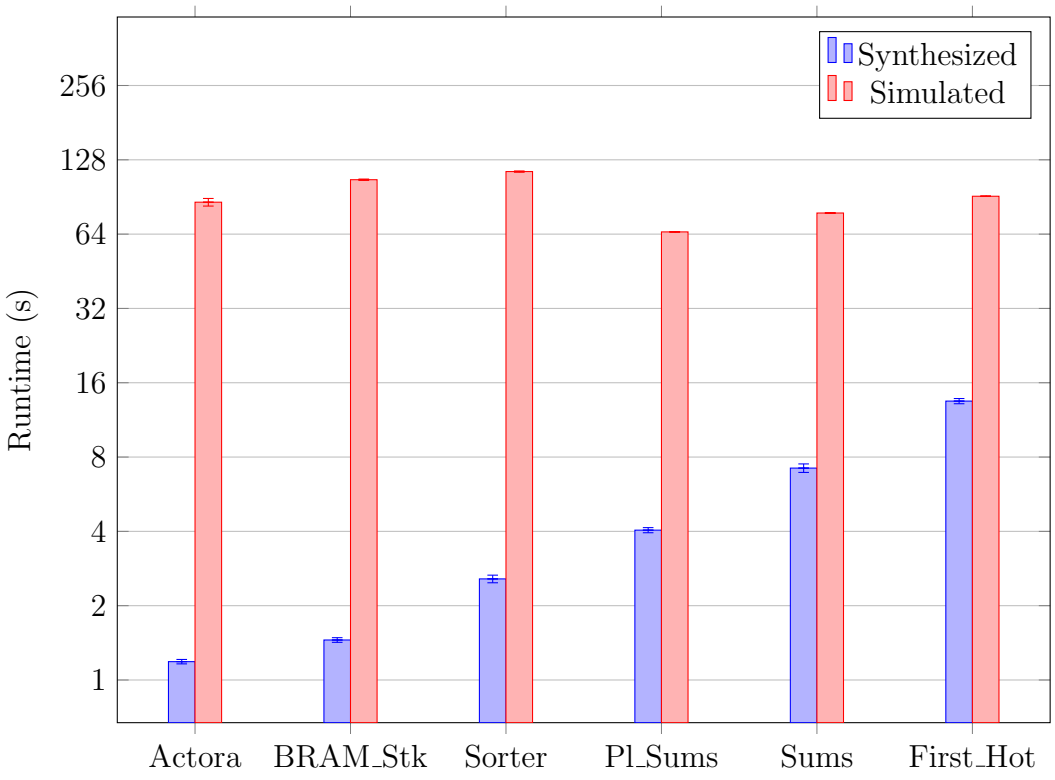


**Figure 4.5:** Runtime comparison of simulated and synthesized versions. One half the runtime gives a one bit advantage.

The runtimes of all the modules are plotted in Figure 4.5. For example, the 3.43 bits advantage from running the Sums properties natively can be seen — it is the difference between the two plots. The CPU example had an insignificant runtime at depth 1 (with a 19.5 bit requirement) but was infeasible to simulate to depth 2 (37.3 bit requirement) and thus is not included in this plot.

In general simulation will be fast enough for most combinatorial logic modules, using only pure properties. Such modules should always be parameterizable in the data width and thus can be instantiated in a small enough form for testing. For example, when verifying an adder, it generally suffices to test all possible 8 bit wide inputs for correctness and assume that this implies a scaled up version will also be correct.

Such reasoning only fails for highly parallel modules, which inherently lend to a hardware implementation. One example which I have implemented is the sorter network, but even here, relatively long lists can successfully be simulated due to the zero-one principle. For more complex modules where such a simplifying principle does not apply, simulation may be too slow.

A table of the clock frequencies simulation and synthesis can achieve for each example is given in Figure 4.6. The advantage of running the test bench natively on an FPGA compared to simulation is also given. For example, as I was able to simulate the

| Example | Synthesized (MHz) Mean | Synthesized (MHz) Std Dev | Simulated (MHz) Mean | Simulated (MHz) Std Dev | Advantage (bits) Mean | Advantage (bits) Std Dev |
|---|---|---|---|---|---|---|
| Actora | 89.86 | 1.85 | 1.24 | 0.04 | 6.18 | 0.07 |
| CPU | 127.68 | 3.1 | 1.76 | 0.01 | 6.18 | 0.04 |
| BRAM Stack | 160.76 | 3.41 | 2.2 | 0.01 | 6.19 | 0.03 |
| Sorter | 26.17 | 0.91 | 0.59 | $< 0.01$ | 5.48 | 0.05 |
| Parallel Sums | 132.79 | 3.16 | 8.22 | 0.08 | 4.01 | 0.04 |
| Sums | 149.01 | 5.95 | 13.78 | 0.03 | 3.43 | 0.06 |
| First Hot | 79.71 | 1.93 | 11.79 | 0.02 | 2.76 | 0.03 |

**Figure 4.6:** Testing speeds on FPGA compared to simulation. The synthesized frequency is the fmax value and the simulated frequency is calculated by dividing the clock cycles simulated by simulation time. The advantage represents the number of bits gained by running the design natively compared to simulation.

Sorter test bench on lists of length 26 in a reasonable amount of time, by executing on an FPGA one could increase this value by 5.48 (advantage bits) and achieve the same runtime — meaning lists of length 31–32 would be comparable. For most modules this is a safe assumption; however, increasing the sorting network width will slightly reduce fmax. Thus for this specific example, a list length of 31 can be run natively in the same time as simulating all lists of length 26.

Due to the complexity of stateful modules, the simulated frequency of these examples is generally lower than that of combinatorial modules. On the other hand, the synthesized test bench is not slowed down by complexity, only by critical path. Therefore synthesis yields more of an advantage in such cases. This trend is depicted as advantage in Figure 4.6. The only possible exception could be the sorting network, for reasons prior explained. Stateful modules will also generally have higher bit requirements, since impure bit requirements are multiplied by maximum depth (equation 3.4). For these reasons, synthesizing the test benches of stateful modules rather than simulating them is much more advantageous than doing the same for combinatorial modules. For example, I was unable to simulate the CPU test bench to depth 2 as this would have taken an estimated 27 hours in simulation — the same testing could be run on an FPGA in under 23 minutes.

In some cases these bit requirements can far exceed even the base bits given by synthesis. In this situation the simple solution is to decrease the maximum testing depth or width of `Forall` assertions. Yet, when this is unsatisfactory one must look for other ways to try and achieve more bits of performance. A potential solution comes in to form of test bench duplication.

From analysing the area usage of all the designs, it is clear that the number of Adaptive Logic Modules (ALMs) used would be the limiting factor. The fraction of

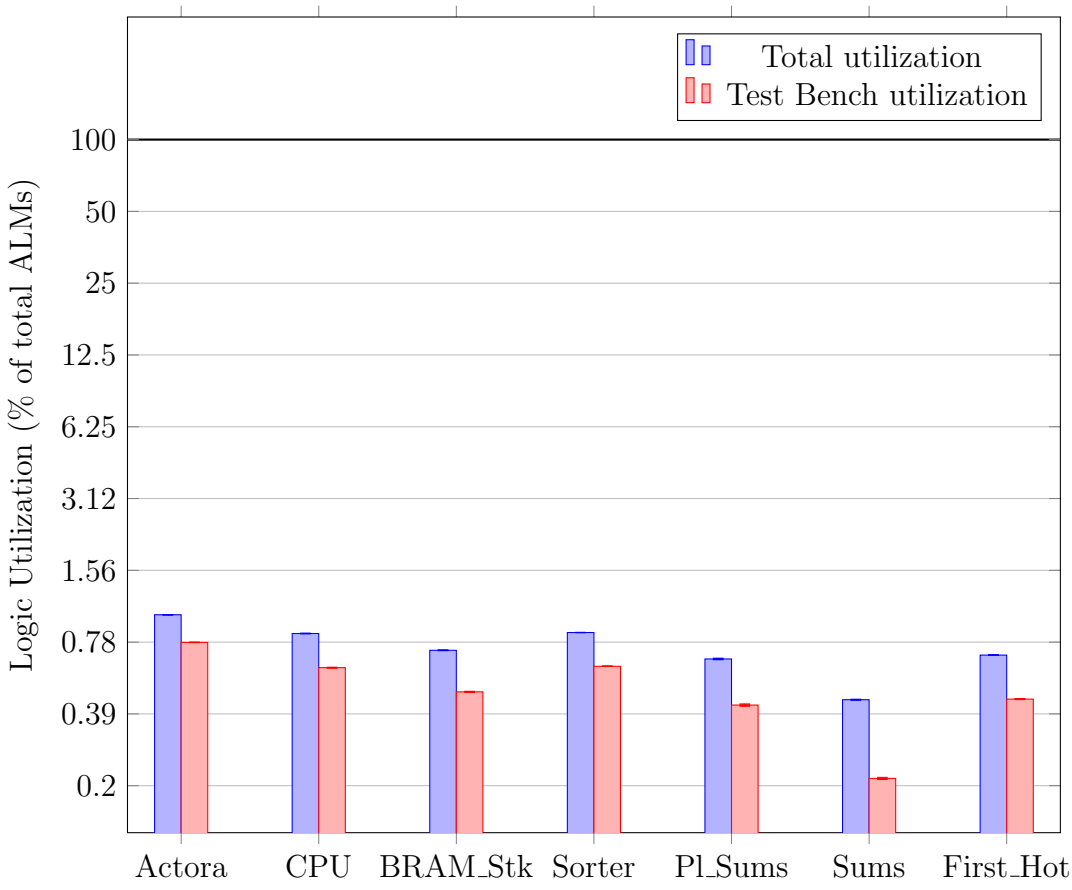


**Figure 4.7:** FPGA area used out of total

total BRAM used was under 0.02% for the module with the highest BRAM usage; the BRAM stack example, compared to an ALM usage of around 0.5%. The ALMs used used in each example is plotted in Figure 4.7. Extra ALMs were taken up by the UART interface component, thus ALM utilization of the test bench itself is also given. The black line shows the maximum ALMs available on the Cyclone V FPGA used.

It is easy to calculate the potential bits which could be gained by duplicating the test bench — each duplication (from $n$ clones to $2n$) gives one bit of advantage. For example the BRAM stack test bench, which utilises 0.5% of the total ALMs, could gain up to an extra 7.64 bits when $2^{7.64} \approx 200$ copies are executed in parallel. In this way one can achieve an advantage of $7.64 + 6.19 = 13.83$ bits over simulation — for the BRAM stack test bench one second of *duplicated* FPGA testing would take over 4 hours to simulate.

BlarneyCheck currently does not support fully parametrizable duplication, but a more ad-hoc version is possible with the use of custom generators. Such a use is demonstrated by the Parallel Sums example, which duplicates the normal Sums properties and thus has a bit requirement of 29 rather than 30. Figure 4.7 depicts this

difference in terms of ALMs.

In summary, simulation is sufficient for most use cases and can easily replace unit tests to provide more confidence in the module under test. For stateful modules which are to be tested to large depths, running the test bench natively on an FPGA can provide a sufficient reduction in runtime to make testing feasible. However, when even larger test cases are necessary one may have to turn to parallel testing.

### 4.4.1 Comparing BlarneyCheck and BlueCheck

Initially it seemed that this project would have far more in common with SmallCheck than BlueCheck. However, it is clear that the difference imposed by testing hardware modules is far greater than that between random and exhaustive testing. Therefore it is fitting to draw a comparison to BlueCheck; some of the differences and similarities are depicted in Figure 4.8. Apart from the obvious approach to test case generation, the biggest difference is that when counter examples are found they are guaranteed to be minimal. This helps uncover the cause of bugs more easily as the designer can be certain that each step in the failing sequence is necessary. Additionally small modules can be tested exhaustively, meaning that their correctness is guaranteed.

In order to hit the high bit requirements of exhaustively testing some modules, parallel testing can be achieved through the use of custom generators. However, due to Haskell restrictions on instance definitions, custom generators are only allowed for different data types. Thus, any value from a custom generator must be encapsulated within a user defined data type — this is only a syntactic flaw.

| | My Project | BlueCheck |
|---|---|---|
| Random testing | ✗ | ✓ |
| Exhaustive testing | ✓ | ✗ |
| Sequential testing | ✓ | ✓ |
| Hybrid parallel/sequential testing | Ad-hoc | ✗ |
| Combinatorial properties | ✓ | ✓ |
| Custom generators | ✓ | ✓ |
| Overridable generators | ✗ | ✓ |
| Guaranteed minimal counter example | ✓ | ✗ |
| Proves correctness | ✓ | ✗ |
| Quantifying over lists | ✓ | ✓ |
| Stateful properties | ✓ | ✓ |
| On-FPGA testing | ✓ | ✓ |

**Figure 4.8:** Comparison to BlueCheck

# Chapter 5

# Conclusion

I have achieved all of the goals I set for the project by creating a library which will exhaustively test arbitrary properties specified in the Blarney HDL. Moreover, I fully implemented a novel extension by extending the expressiveness of the property language to enable testing of stateful logic modules. Additionally, a second extension was partially explored in which I demonstrated that my BlarneyCheck library is also capable of systematic random testing.

To evaluate my work I created a comprehensive suite of test examples which demonstrate the full capabilities of my library. These were then both rigorously simulated and synthesized for analysis. I found that the simulation was only useful for verifying small properties or simple modules; however, this generally proved to be enough to find most bugs and was much easier to run than a full synthesis. If simulation is infeasible then running natively on an FPGA can provide a speedup of multiple orders of magnitude, this is only practical for large examples due to the overhead of synthesis. Such a speedup was particularly evident when testing modules which are more naturally expressed in hardware, such as sorting networks or stateful modules.

Overall, BlarneyCheck improves on previous work by providing more confidence in the correctness of modules which pass testing. Alternatively, any counter-examples found are guaranteed to be minimal, making it significantly easier to discover the root cause of the failure. Multiple tools are provided to help reduce the required search space to a feasible size for exhaustive testing. When used on a pre-existing project my library managed to find interesting and undocumented failing edge cases.

## 5.1 Further work

I have not noticed any aspects of my work that could be significantly improved with more time. Nevertheless, whilst working on the project I discovered multiple new areas which would be interesting to explore.

- *More expressive generators* — There were two main types of generators that I experimented with during the project. The initial one required the user to define a standalone module with it's own internal state, but this proved far too complex to expose to the end user. Thus, I opted for a simpler approach where generators only provide functions to move from one state to the next. If the former could be made more user friendly I would expect more custom types could be exhaustively generated.

- *Random yet exhaustive testing for stateful properties* — To demonstrate the expressiveness of custom generators I created one to generate all values of any given bit-width in a pseudo-random order. Though this does not provide fully random checking for stateful properties as values are still generated exhaustively one by one. To fully achieve random testing a different approach would have to be taken.

- *More extensive use* — To fully evaluate the usefulness of such a library as this, it must be used to complement the entire development process of other projects. Thus it would end up being used to test more complex examples such as fully featured classical processor designs, facilitating further development of this library.

- *Parallel verification* — Currently each property to be checked is instantiated once and incremented once per clock cycle. For simple properties this potentially takes up very little area on an FPGA. Thus a replication 'slider' could be introduced to split up the test cases evenly between multiple replicated test benches, trading off a higher area usage for reduced testing time. (Custom generators make this non-trivial)

- *Extended property language* — Probably the most interesting extension would be to add new property constructors. This could include an *Exists* statement, similarly to what SmallCheck implements, or a *ForallFunction* to exhaustively generate arbitrary functions (hardware modules), implemented simply by generating LUTs.

# Bibliography


[1] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 35, p. 268–279, Sept. 2000.

[2] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values," in *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, (New York, NY, USA), p. 37–48, Association for Computing Machinery, 2008.

[3] M. Naylor and S. Moore, "A generic synthesisable test bench," *MEMOCODE*, 08 2015.

[4] M. Naylor and A. Joannou, "Blarney." `https://github.com/mn416/blarney`, 2020.

[5] A. Gill and C. Runciman, "Haskell program coverage," in *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2007.

[6] K. Claessen, "Embedded languages for describing and verifying hardware," tech. rep., Chalmers University of Technology and Göteborg University, 2001.

[7] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," *SIGPLAN Not.*, vol. 34, p. 174–184, Sept. 1998.

[8] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[10] M. Naylor, "Actora." `https://github.com/mn416/actora`, 2020.

[11] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," 1973.

# Appendix A

# Project Proposal

Part II Project Proposal

## Property-based Testing for Hardware

October 2019

**Project Originator:** Matthew Naylor

**Project Supervisor:** Matthew Naylor

**Directors of Studies:** Simon Moore and Hatice Gunes

**Overseers:** Pietro Lió and Robert Mullins

# Introduction and Description of the Work

Test benches are common when developing hardware and so many techniques have been developed for automatic testing and generation of test cases. Property-based testing requires the designer to specify a set of invariants for the module being tested, this allows automatic testing to check that these invariants always hold, reporting back if that is the case. This is a useful and now commonplace idea in automatic testing.

Property-based testing has become popular in the software community. Both **QuickCheck** and **SmallCheck** are existing testing libraries that encourage lightweight formal specification of software through the reward of automatic testing and small counter-examples. Recent work has successfully applied the idea to hardware development in the Bluespec HDL.

Hardware verification tools are often commercial products and expensive, so the open-source community needs new approaches. This project will continue the work of **BlueCheck**, but explore some new avenues. Namely we will use an open-source Bluespec-like HDL developed here in the CL, additionally we will use bounded exhaustive testing, which can can find minimal counterexamples without the need for shrink steps.

Here is an example of specifying and testing the firstHot function from the BlueCheck paper. This function returns a bit-string in which only the least significant non-zero bit of the input bit-string is set:

```
// FirstHot implementation to test
function Bit#(n) firstHot(Bit#(n) x) = x & (~x+1);

// The defining properties that any First Hot function must meet:
// Exactly one output bit is set if nonzero, otherwise no bits set
function Bool prop_OneIsHot(Bit#(8) x) =
  countOnes(firstHot(x)) == (x == 0 ? 0 : 1);

// Hot bit in output is also hot in input
function Bool prop_HotBitCommon(Bit#(8) x) =
  (x & firstHot(x)) == firstHot(x);

// No less-significant hot bits in input than that set in output
function Bool prop_HotBitFirst(Bit#(8) x) =
    (x & (firstHot(x)-1)) == 0;
```

# Starting Point

I have basic prior knowledge, from taking the **Computer Design** course in **Part IB** and the **Digital Electronics** course in **Part IA** including the practicals for both. Therefore an important part of the project will be learning to use the required tools.

# Substance and Structure of the Project

The aim of the project is to build a Haskell library for automatic testing of modules written in Blarney. Primarily I will draw inspiration from SmallCheck and apply it to a HDL. As such a method using bounded exhaustive testing will be implemented. This gives a clear demarcation between tested and untested cases, and also guarantees finding small counterexamples. The work consists of three main parts, described in the success criteria.

The project will likely be expanded to achieve some of the additional points mentioned in the next section, but which ones are chosen will depend on the effectiveness of the initial implementation. Further options may be discovered when researching this topic in depth and also implemented.

# Success Criteria

The following should be achieved:

- Implement a Haskell library for automatic testing of combinatorial circuits written in Blarney
- Devise a suite of example properties and buggy circuits, to evaluate the effectiveness of the library
- Measure the proportion of bugs reported and the size of counter-examples found
- Synthesise a test bench to an FPGA and compare speeds against a simulated test bench

Finally there are many further possibilities beyond the success criteria for this project:

**Sequential logic** modules could be supported, testing a bounded sequence of inputs

**Random testing** similar to QuickCheck and BlueCheck

**Other verification methods** (such as an SMT solver) could be created for Blarney and then compared to the bounded exhaustive testing method

# Timetable and Milestones

## 25th October to 21st November

Research property-based testing for software and also hardware, making sure to fully understand the related work. Setup and test the SmallCheck/QuickCheck libraries for Haskell, to better understand how they work and to improve skills with Haskell. Learn to use Blarney, building some example circuits as a learning exercise.

Milestones: Small Blarney modules of combinatorial logic implemented. Structure of testing framework and type-based generators designed.

## 22nd November to 5th December

Setup BlueCheck and Bluespec to test synthesizing test benches to an FPGA. Decide with the help of Matt on final design of the testing library, with further literature study if necessary. Create the enumerative generators.

Milestones: Structure necessary to meet the success criteria laid out and ready to work on my own over the Christmas vacation.

## 6th to 26th December (Christmas vacation)

Complete success criteria, namely: Implement a Haskell library for automatic testing of combinatorial circuits written in Blarney. If any problems arise can use second half of Christmas vacation to finish the work, at the cost of fewer additional features. Otherwise if finished early start work on Dissertation.

Milestones: Code satisfies the first point of the success criteria. Introduction chapter of Dissertation mostly complete.

## 27th December to 16th January (Christmas vacation)

Start building additional features as able, with priority for synthesizing test benches to an FPGA. Write Preparation chapter of Dissertation and start work on Implementation chapter.

Milestones: Synthesized test bench runs on FPGA. First 2 chapters of Dissertation mostly complete and structure of Implementation chapter laid out. Some of the proposed additional features complete.

## 17th January to 27th February

Create progress report and presentation. Run the library on the example buggy circuits in preparation for Evaluation chapter. Start of lent term is very intense so use the long time at low intensity to make small improvements and find any bugs.

Milestones: Progress report and presentation done. Test data for Evaluation generated.

## 28th February to 12th March

Complete all additional features and clean up code. Project should be stable at this point, with all changes being only minor after this point.

Milestones: Project is complete and stable to allow writing Dissertation.

## 13th March to 23rd April (Easter vacation)

Finish up the Introduction and Preparation chapters and write Implementation chapter. Make only small crucial changes to code (e.g. bug fixes)

Milestones: First 3 chapters of Dissertation complete. Codebase finalized, ready for submission.

## 13th March to 23rd April (Easter vacation)

Write Evaluation and Conclusions chapters, running any additional tests to gather evaluation data if necessary. I can use this time to do additional work on the project if Timetable has been moved back, otherwise for revision for final exams.

Milestones: Dissertation finished, ready for submission if necessary.

## 24th April to 7th May

Review whole project, check the Dissertation, and spend a final few days on whatever is in greatest need of attention.

Milestones: Dissertation is polished. Submission of Dissertation.