A Generic Synthesisable Test Bench

Matthew Naylor and Simon Moore
Computer Laboratory, University of Cambridge, UK
{matthew.naylor, simon.moore}@cl.cam.ac.uk

***Abstract*—Writing test benches is one of the most frequently-performed tasks in the hardware development process. The ability to *reuse* common test bench features is therefore key to productivity. In this paper, we present a *generic* test bench, parameterised by a specification of correctness, which can be used to test *any* design. Our test bench provides several important features, including automatic test-sequence generation and shrinking of counter-examples, and is fully synthesisable, allowing rigorous testing on FPGA as well as in simulation. The approach is easy to use, cheap to implement, and encourages the formal specification of hardware components through the reward of automatic testing and simple failure cases.**

I. Introduction

Test benches are perhaps the most commonly used method of verifying hardware correctness, yet they are often written 'ad-hoc' with little or no code shared between the test bench of one component and that of another. There are however a number of desirable functionalities common to almost any test bench – e.g. generating test-sequences, monitoring coverage, finding simple failures, and reporting outcomes – many of which can be tricky or time-consuming to get right. If such features are abstracted out in an easily-reusable form – e.g. in a highly-generic test bench, as we propose in this paper – then testing becomes not only easier to do, but also more effective.

A key step in developing any test bench is deciding which behaviours of the implementation should be allowed and which should not. To use our test bench, we require the developer to provide an *executable specification* that defines the validity of any conceivable input-output combination. This allows the testing process to be *automated*. We adopt the idea of writing specifications in the same language as the implementation [1]. Compared to full formal specification, this has pros and cons: on the one hand, no new specification language needs to be learned; on the other, only properties expressible within the HDL can be tested.

Verification of stateful hardware components typically involves the use of *test-sequences*, i.e. sequences of inputs that vary over time. In our experience, long test-sequences that lead to failures are extremely difficult to diagnose – a failure can be a consequence not only of the latest input, but of all past inputs too. This motivates two features in our test bench for automatically finding simple failures: *iterative deepening* and *shrinking*. Iterative deepening involves the generation of test-sequences whose sizes gradually increase over time, with the aim of finding simple failures first. (Between each test-sequence, the design under test is reset.) In contrast, shrinking involves taking a long failing sequence and attempting to shorten it by repeatedly omitting possibly-unneeded elements. We have found *both* techniques to work well in practice, and especially when used in combination.

Reporting failing test-sequences to the user in a hand-written test bench usually involves the use of simulation-only `$display` statements. When running on FPGA, these statements are simply ignored and the developer must explicitly write code to transfer outcomes to a host PC, e.g. over a UART. Additional code must be written if they wish to accurately *replay* the failing sequence in simulation. A similar situation arises when generating sample input data using `$random` statements, which are not synthesisable either. All this means that hand-written test benches tend not to run on FPGA, which is unfortunate since testing can be far more thorough there. Our generic test bench solves each of these issues once and for all, encouraging the practice of on-FPGA testing.

To generate sample test data we use *random testing*. Although effective, care must be taken to avoid certain pitfalls [1]. In particular, random testing is sometimes insufficient to generate a good percentage of interesting inputs, especially when these inputs must satisfy strict invariants. To address this concern, we provide support for classifying the generated test data to determine where test coverage is poor, and allow the default generator for any data type to be customised so as to bias the distribution of test data in the desired direction.

Our generic test bench is implemented simply as a library module in the Bluespec HDL and is called *BlueCheck*, after *QuickCheck* [1], an influential software testing tool on which BlueCheck is based. BlueCheck is available open-source from:

`http://github.com/CTSRD-CHERI/bluecheck`

Although we will show that Bluespec turns out to be an excellent language to implement a generic test bench, we believe that the core ideas can be usefully applied to other HDLs too. We begin this paper by presenting the design and implementation of BlueCheck, highlighting the ideas in an example-driven manner (§II). After that, we give our experiences of using BlueCheck in a real hardware project, specifically the shared memory subsystem of the BERI multi-processor [2] (§III). We then discuss HDL features that assist the development of a generic test bench (§IV), and present related work, noting in particular how BlueCheck compares with other recent testing tools and initiatives (§V and §VI).

While the hardware verification community continues to make great strides in the effective use of formal methods to produce high-quality designs, the value of formally specifying hardware components is still not as widely appreciated as it could be. Not only does writing a specification help the developer to better understand the problem at hand, but it gives them an oracle to rigorously test against, as well as the ability to search for simple failure cases during debugging. The aim of this paper is to illustrate these benefits in the context of hardware design, and to demonstrate how they can be achieved within a modern HDL.

II. BlueCheck

A. An introductory example

Let us begin with a simple example. Suppose we require an HDL function `firstHot`, which given a bit-string x returns a similar bit-string in which only the least significant non-zero bit of x is set. In Bluespec, we can define it as:

```
function Bit#(n) firstHot(Bit#(n) x) =
  x & (~x + 1);
```

This function returns the bitwise conjunction of the input and the 2's complement of the input. It is not immediately obvious that this works! So we will test it by first writing a specification and then using BlueCheck to verify that `firstHot` does indeed meet that specification.

To specify the correct behaviour of `firstHot`, we identify two important properties which it must satisfy. First, the output must be a bit-string in which exactly one bit is set, unless the input is all zeros, in which case no bits should be set. We might formulate this as:

$$\texttt{countOnes(firstHot}(x)\texttt{)} \quad = \quad \begin{cases} 0 & \textit{if } x = 0 \\ 1 & \textit{otherwise} \end{cases}$$

Here, `countOnes` counts the number of non-zero bits in a string, and happens to be provided by Bluespec's standard libraries. Another important property of `firstHot` is that the hot bit in the output must also be hot in the input:

$$x \texttt{ \& firstHot}(x) \quad = \quad \texttt{firstHot}(x)$$

Often when using `firstHot`, the programmer will rely on these two properties alone – e.g. when they care only that *some* hot bit in the input is isolated, not necessarily the first one. However, for completeness, a third property might be stated, namely that there are no hot bits in the input that are less significant than the hot bit in the output. We encourage readers to define this third property themselves. (*Hint:* it can be expressed concisely using only standard arithmetic and bitwise operators. A possible answer is in a footnote[1].)

Now that we have decided on the properties of `firstHot`, it is straightforward to express them using standard Bluespec functions in which any universally quantified variables, such as x in the above properties, are taken as function arguments:

```
0  module [Specification] firstHotSpec ();
1    // One bit in the output is hot
2    function Bool oneHot(Bit#(4) x) =
3      countOnes(firstHot(x)) == (x == 0 ? 0 : 1);
4
5    // The hot bit is common to input and output
6    function Bool hotCommon(Bit#(4) x) =
7      (x & firstHot(x)) == firstHot(x);
8
9    // Register properties with BlueCheck
10   prop("oneHot"   , oneHot);
11   prop("hotCommon", hotCommon);
12 endmodule
```

In the above specification, identifiers that are exported by the BlueCheck library are **underlined**. On lines 2 and 6, even though the two properties are in principle agnostic to the size of bit-strings, we have had to specify a fixed size bit-string – we choose `Bit#(4)` – on which the properties are to be tested. On line 0, we use a feature of Bluespec which allows a module to collect information of some type specified in square brackets after the `module` keyword. Here, we are collecting the specification to be tested: on lines 10 and 11 we register two properties via calls to `prop`. In this example, each call to `prop` is passed a property with a single argument of type `Bit#(4)`; in general however, `prop` is an overloaded function and may be passed a property which takes any number of arguments, each of any type.

Once a specification has been written, it can be passed to BlueCheck:

```
module firstHotChecker ();
  blueCheck(firstHotSpec);
endmodule
```

Now, when `firstHotChecker` is simulated as the top-level module, we obtain the following output.

```
OK: passed 1000 iterations
```

The `blueCheck` function has instantiated a test bench in which each property is applied to random inputs on each clock-cycle, for a thousand iterations, and the properties never returned false. Had we mistakenly defined our `firstHot` function as

```
function Bit#(n) firstHot(Bit#(n) x) =
  x & ~(x + 1);
```

then we would obtain the output:

```
oneHot(15)
Property does not hold
```

As expected, a counter-example is reported: the `oneHot` property no longer holds when x is `15`. This test bench can also be synthesised and run on FPGA for more rapid testing. We will give more details about this later, but for now we turn our attention to the testing of stateful circuits which pose a greater challenge than purely combinatorial ones such as `firstHot`.

B. Testing stateful circuits

As an example of a simple but interesting stateful circuit, let us consider a stack implementation providing all the usual operations captured by the following interface.

```
// A stack of 2^n elements of type t
interface Stack#(type n, type t);
  method Action push(t x);
  method Action pop;
  method Bool isEmpty;
  method t top;
  method Action clear;
endinterface
```

In Bluespec, an `Action` method is simply one that may have an implicit side-effect, e.g. by mutating some internal state that is not visible to the caller. In contrast, a non-`Action` method can only read such internal state, not modify it.

Now we will test an implementation of the above interface called `mkBRAMStack`. Rather than listing the implementation here, we simply note its two main features: (1) it uses a block RAM to store the elements of the stack, allowing large stack sizes to be supported; and (2) it caches the top stack element

[1] Possible answer: x `& (firstHot(`x`) - 1) = 0`

in a register to reduce propagation delay on the output of the `top` method. To begin, we present a basic unit test for the implementation – this will serve as an illustration of Bluespec method calls as well as a commonly-used testing method:

```
0  module stackUnitTest ();
1    // Create 256-element stack of 4-bit values
2    Stack#(8, Bit#(4)) stk <- mkBRAMStack;
3
4    // Unit test, written as a sequence of calls
5    Stmt test =
6      seq
7        stk.push(1);
8        stk.push(2);
9        if (stk.top != 2) $display("Failed");
10       stk.pop;
11       if (stk.top != 1) $display("Failed");
12     endseq;
13
14   // Generate a state machine from the sequence
15   // (with one state per line of the seq block)
16   mkAutoFSM(test);
17 endmodule
```

If we simulate the above module, we find that no failure messages are displayed and thus conclude that the test passes. Although reassuring, the result is not compelling: if we consider all 5-element sequences of operations on a stack containing 4-bits values, this test has only checked one out of over 3 million possibilities. Furthermore, it does not consider parallel behaviour, i.e. the possibility that several stack methods are called in the same clock-cycle.

For greater confidence, we now write a specification for `mkBRAMStack` and then use BlueCheck to verify it. Various approaches may be taken to specify a stack: for now, we define a golden model and assert equivalence between the golden model and the implementation; later we will use an algebraic specification. Our golden model is called `mkRegStack` and implements the entire stack using registers: it ignores low-level issues such as logic usage and block RAM access protocols, and is more obviously correct than `mkBRAMStack`. Using BlueCheck, the specification looks as follows.

```
0  module [Specification] stackSpec ();
1    // Implementation instance
2    Stack#(8, Bit#(4)) imp <- mkBRAMStack;
3
4    // Golden model instance
5    Stack#(8, Bit#(4)) model <- mkRegStack;
6
7    equiv("push"   , model.push   , imp.push);
8    equiv("pop"    , model.pop    , imp.pop);
9    equiv("isEmpty", model.isEmpty, imp.isEmpty);
10   equiv("top"    , model.top    , imp.top);
11 endmodule
```

We have specified that each method in the golden model is equivalent to the corresponding method in the implementation. Once again, we pass the specification to BlueCheck:

```
module stackChecker ();
  blueCheck(stackSpec);
endmodule
```

Simulating `stackChecker` as the top-level module, we get:

```
push(13)
pop
push(6)
... 13 method calls elided ...
pop
pop
top failed: 6 v 1
```

As it turns out, there is a bug: we see a counter-example in the form of a sequence of method calls ending in a call to `top` which returns `6` in the golden model but `1` in the implementation. The failing sequence is 19 elements long in total, but most of it is elided for space reasons. To diagnose the bug, we would ideally like a smaller failing sequence. However, before looking at that, it is helpful to know a little more about how the test bench actually works.

C. A basic testing strategy

It is useful to distinguish properties that mutate state (*impure* properties) from those that don't (*pure* properties). For example, in the stack specification above, the `"push"` property on line 7 is impure whereas the `"isEmpty"` property on line 9 is pure.

The BlueCheck test bench is a state machine containing one state for each impure property that has been specified, as well as a single no-op state. The current state of the machine is taken from the output of a pseudo-random generator, leading to a random sequence of property-invocations, and the value of each argument to each property is also generated at random. Since pure properties do not conflict with other properties, they are always invoked, regardless of the current state of the checker, even when in the no-op state. While calls to pure properties are only displayed by BlueCheck when they fail, calls to impure properties are always displayed because they may contribute to a failure.

It is important to note that just because a property is invoked does not mean it actually fires. In Bluespec, a method may have an implicit condition that must be satisfied for it to fire, e.g. the stack `pop` method may be conditioned on the stack being non-empty. So the randomly-changing state machine may well generate strange sequences, such as popping from an empty stack, but such sequences will be naturally handled by a property's implicit conditions: if a property cannot fire, BlueCheck will simply move on to a new random state on the next clock-cycle.

Another convenient feature of Bluespec that we exploit is atomicity: when processing a specification containing an impure `equiv` property, BlueCheck will place both sides of the equivalence in the same atomic rule. This means, for example, that the `"push"` property on line 7 of `stackSpec` will only fire if the `push` method can fire in both the model *and* in the implementation, otherwise the two stacks would become out of sync. This is clearly important for correct equivalence checking.

D. A better testing strategy

A big limitation of the above testing strategy is that it only has the ability to check a single, albeit extremely long, test sequence. To check many different test-sequences, we require the ability to reset the circuit under test. To illustrate, let us return to our stack example. With two small tweaks we can make a resettable version of `stackSpec` called `stackSpecR`.

First, we modify line 0 to take an external reset signal as a module argument:

```
module [Specification] stackSpecR#(Reset r) ();
```

And second, we modify lines 2 and 5 to to pass the reset signal down to the stack implementation and golden model respectively. For example, line 2 now reads:

```
Stack#(8, Bit#(4)) imp <-
  mkBRAMStack(reset_by r);
```

Given a resettable specification, BlueCheck uses a more elaborate testing strategy which we refer to as *iterative deepening* (ID). The idea is to start by generating test-sequences of some initial depth (i.e. length), and gradually increase the depth over time. Between each test sequence, the circuit under test is reset back to its initial state. Finally, if a failure is found, the test bench will attempt to *shrink* the failing sequence. This shrinking strategy operates as follows:

(1) Omit an element of the failing test sequence.
(2) Replay the test-sequence, and if it no longer fails, reinsert the omitted element.
(3) If all elements of the failing test sequence have been considered for omission, then halt.
(4) Otherwise, go back to step 1.

Notice that shrinking requires the ability to replay a test-sequence. This is achieved by resetting the circuit under test, restoring the seeds of the random generators to the values they had at the beginning of the test, and invoking each non-omitted element of the sequence on the same clock cycle that it was invoked in the original failure. If the omission of an element from the test-sequence causes later elements not to fire, then naturally these will be removed too. New elements that did not appear in the original sequence are never introduced.

We can now pass our new resettable specification to BlueCheck and it will use the new testing strategy.

```
module stackCheckerID ();
  blueCheckID(stackSpecR);
endmodule
```

Simulating it gives the following output.

```
=== Depth 20, Test 1/10000 ===
Saving state to 'State.txt'
39: push(1)
40: push(7)
43: push(10)
44: pop
46: pop
47: top failed: 1 v 7
Continue searching?
```

Now we obtain a fairly short failing sequence, consisting of just 6 elements. In fact, this is a minimal counter-example: it is not possible to expose the bug using fewer calls. In general however, minimal counter-examples are not guaranteed, so BlueCheck allows the user to continue testing with the possibility of finding a simpler one. Note that each property call is now preceded by a timestamp – the clock-cycle on which the call was made. The fact that the times are not perfectly contiguous is due to one of three possible reasons: (1) the no-op state was visited; (2) the chosen property did not fire; or (3) shrinking has removed some of the original calls.

E. Resettable specifications

BlueCheck's iterative deepening and shrinking procedures both rely on the ability to reset the design under test back to a well-defined initial state. Most primitive Bluespec components, such as registers and FIFOs, are resettable out-of-the-box and hence writing such resettable specifications usually requires no special attention from the developer. However, one notable exception is Bluespec's block RAM component: its contents are not reinitialised on reset. If a design makes any assumption about the initial contents of a block RAM then that block RAM should be initialised explicitly. We note that proper resetting of modules in this way is considered good practice not just for BlueCheck testing, but also for any component that may be placed on an SoC bus.

F. Saving and replaying counter-examples

Before running each test-sequence, BlueCheck saves the state of all its random generators and if the test-sequence leads to a failure then this state is dumped either to a file (in simulation) or over a UART (on FPGA). For example, we can see in the above counter-example that the state has been dumped to the file `State.txt`. Any counter-example can be replayed at a later time by passing `+replay` as an argument to the simulator, which will cause `State.txt` to be loaded by the test bench. This allows counter-examples to be saved as regression tests, and replayed in isolation with additional debug/tracing options enabled, but the big attraction is that counter-examples found on FPGA can be easily replayed in simulation.

While in many cases the replay feature works well, it is not always true that a failure found on FPGA will correspond to a failure in simulation: the design under test may use a hardware component whose behaviour is not perfectly modelled in simulation. In such cases, BlueCheck at least allows the failing test-sequence to be *viewed* by passing `+view` to the simulator. To achieve this, BlueCheck not only dumps the state of the random generators when a failure is found, but also the clock-cycle on which each property fired. This is sufficient information to deduce the failing sequence that occurred.

It is straightforward to run BlueCheck on FPGA. When targeting Altera FPGAs, BlueCheck provides a single memory-mapped master interface that can be connected directly to Altera's standard JTAG UART component. If the output of the UART is redirected to a file `State.txt` on the host PC then the results of testing can be viewed in the simulator. Using BlueCheck to test `stackSpecR` on a DE4 FPGA development board, we are able to test 1.8M 20-element test-sequences per second, running at 100MHz. This compares to 8K sequences per second in simulation.

G. The value of shrinking

It is useful to ask whether the shrinking procedure is really necessary, or whether the simpler iterative deepening strategy alone is enough to find small counter-examples. The problem with using iterative deepening alone is that it raises a tricky question: how rapidly should the depth be increased? If increased too quickly, we lower the chance of finding a small failure. And if increased too slowly, it may take too long to find any failure. Let us illustrate using the `stackSpecR` example.

Tests-per-depth	Avg. time to first failure (cycles)	Avg. size of first failure
10	1704	10
50	4468	7
100	7409	6
500	27388	5

Fig. 1. Using iterative deepening (without shrinking) to test the stack example, with an initial depth of one. The depth is incremented by one after every *tests-per-depth* test-sequences. The results are averaged over 100 runs, each run using a different random seed.

Figure 1 shows the effect of varying the number of tests-per-depth on the average size of the first-counter example found, and the time taken to find it. The ideal trade-off depends on the answer to a question which is unknown in advance: how readily is a failure observed at each depth?

Using shrinking, we avoid these difficulties. Even if we increase the depth rapidly, or choose a large initial depth, small failures may still be found. For example, in the stack example, if we begin testing at depth 10, we require just 931 cycles on average to find a minimal counter-example. This is better than any of the iterative deepening results shown in Figure 1. In other words, shrinking can allow *simple* failures to be found *quickly*.

H. Concurrent properties

So far, when talking about testing strategies, we have assumed that impure properties conflict with each other and cannot run in the same clock-cycle. While this is a safe default assumption, it is not always true. For example, in Bluespec it is possible to define a stack whose push and pop methods can fire simultaneously, effectively replacing the top stack element, and it is important to be able to test such behaviour. To specify that our `"push"` and `"pop"` properties on lines 7 and 8 of `stackSpec` can run in parallel, we may add the following statement just before line 11.

```
parallel(list("push", "pop"));
```

As a result, an extra state is added to the BlueCheck state machine in which all the properties specified in the list are invoked. In this new state, any subset of the specified properties may fire together: there is no requirement that they all must do so. One slight limitation of this approach is that the shrinking procedure currently only considers the omission of *all* elements that fire on the same cycle, even though only a subset of these may be necessary to reach a failure.

I. Classifiers and frequencies

The reader may have noticed that `stackSpec` does not mention the `clear` method, which removes all elements from a stack. The problem with having calls to `clear` is that the likelihood of constructing stacks with more than a couple of elements becomes rather low. Indeed, if we add the following property to `stackSpec`,

```
equiv("clear", model.clear, imp.clear);
```

then we observe a rise in the number of passing test-sequences before the bug is found. This highlights the importance of monitoring test data when doing random testing, to ensure good coverage of the state space. In response, BlueCheck (like QuickCheck) provides a mechanism for classifying test cases. To illustrate, suppose we wish to monitor the number of "small" stacks being constructed. Adding to `stackSpec`, we can first create a classifier:

```
let small <- mkClassifier("small");
```

A classifier is a function from `Bool` to `Action` and, when called, BlueCheck will internally update true and false counts for that classifier. Now, at the end of each test, we classify a stack as small if it contains two elements or less:

```
post("", small(model.size <= 2));
```

Here we assume that the stack interface has been extended with a `size` method, and we use a BlueCheck routine called `post` which specifies a property to be called at the end of each test-sequence. Now, when testing a corrected implementation of the stack, the test bench reports:

```
OK: passed 20000 test sequences
93% small
```

It is straightforward to bias the test-sequence generator so as to construct larger stacks by replacing lines 7 and 8 of `stackSpec` with:

```
equivf(4, "push", model.push, imp.push);
equivf(2, "pop" , model.pop , imp.pop);
```

We have assigned frequencies to the properties so that, on average, `push` is called twice as often as `pop`, and four times more often than `clear`. The result of testing is now:

```
OK: passed 20000 test sequences
59% small
```

J. Algebraic specification

Another way to test a stack, without the need for a golden model, is to define algebraic properties that define how methods interact with each other [3]. For example, one such property for a stack is that performing a push followed by a pop is equivalent to a no-op. We might express this as:

$$\texttt{stk.push}(x)\ \texttt{;}\ \texttt{stk.pop} \quad = \quad \textit{no-op}$$

We can write a BlueCheck specification to test this law:

```
0  module [Specification] stackSpecAlg ();
1    // Create two instances of implementation
2    Stack#(8, Bit#(4)) s1 <- mkBRAMStack();
3    Stack#(8, Bit#(4)) s2 <- mkBRAMStack();
4
5    // On s1, push x, then pop it
6    function pushPop(x) =
7      seq s1.push(x); s1.pop; endseq;
8
9    // On s2, do nothing
10   function nop(x) = seq endseq;
11
12   equiv("pushPop", pushPop, nop);
13   equiv("push"   , s1.push, s2.push);
14   equiv("pop"    , s1.pop , s2.pop);
15   equiv("top"    , s1.top , s2.top);
16 endmodule
```

Here we create two instances of the stack implementation, and define a property "pushPop" in which one instance performs a push followed by pop and the other performs a no-op. We then assert equivalence between the two instances. Notice that the new property requires a multi-cycle seq block to express that the push and pop calls are performed sequentially in time. Even this one algebraic property is enough to catch the bug in our implementation, without the need for a golden model:

```
=== Depth 20, Test 15/10000 ===
11: push(12)
22: push(2)
23: pushPop(14)
27: pop
28: top failed: 2 v 12
Continue searching?
```

Another important algebraic property concerns the post-condition of the push method: after push(x) is called, the stack should be non-empty and the top element should contain the value x. To express this, we write:

```
function pushPost(stk, x) =
  seq
    stk.push(x);
    ensure(!stk.isEmpty && stk.top == x);
  endseq;
```

This new property can be added to the algebraic specification at line 12. First though, we must bring the ensure function into scope so that BlueCheck can observe the correctness condition (seq blocks do not have return values). At line 4, we write:

```
let ensure <- mkEnsure;
```

Now we can insert the new property at line 16:

```
equiv("pushPost", pushPost(s0), pushPost(s1));
```

Here we make use of partial function application: on each side of the equivalence, we partially apply the property to a different stack.

K. Wedge detection

A *wedge* is a common form of failure in which the design under test locks up and ceases to be productive. Using BlueCheck, any test-sequence that leads to a wedge can be viewed simply by enabling *verbose mode*. This causes BlueCheck to display tests-sequences as they are generated (by default it will only display tests-sequences after they have failed). When a wedge occurs, the simulation will lock up but the sequence causing it will be seen on the output terminal.

This simple approach has two drawbacks: (1) it does not work when the test bench is running on FPGA where $display statements are ignored; and (2) no attempt is made to shrink the failing sequence. To counter these problems, we use a simple wedge detection mechanism which keeps track of the number of consecutive clock-cycles in which no impure property has fired. When this count exceeds a user-defined threshold, the test bench terminates with a wedge failure.

L. Custom generators

By default, BlueCheck generates input data using a pseudo-random generator that is generic across any synthesisable data type. However, sometimes this default generator is not ideal. To illustrate, consider the following type.

```
typedef struct {
   Bit#(n) value;
} OneHot#(type n);
```

The idea is that a value of type OneHot#(n) is a n-element bit-string in which exactly one bit is set, however the default generator will yield values in which any number of bits are set. This motivates the following module which yields a custom generator for one-hot values.

```
0  module [Specification] mkOneHotGen (
1                            Gen#(OneHot#(n))
2                          );
3    // Create generator to yield random indexes
4    Gen#(Bit#(TLog#(n))) index <- mkGen;
5
6    // Method to generate one-hot values
7    method ActionValue#(OneHot#(n)) gen;
8      // Generate random index
9      let i <- index.gen;
10     // Set the bit at the random index
11     let v = 1 << bound(i, valueOf(n)-1);
12     return OneHot { value: v };
13   endmethod
14 endmodule
```

To generate a random one-hot bit-string of length n, we might first generate a random index i of length $log_2(n)$, and then compute $1 << i$. However this method leads to overflow when n is not a power of two and $i \geq n$, hence we actually compute $1 << \texttt{bound}(i, n-1)$. The bound function is used to obtain a value with a given upper bound, and has a cheaper implementation in hardware than modulus divide.

On line 7 we create a generator for random indexes using mkGen, a polymorphic function capable of constructing generators for any synthesisable type. To tell BlueCheck to use our new custom generator for values of type OneHot#(n), instead of the default one, we use the type-class feature of Bluespec and write:

```
instance MkGen#(OneHot#(n));
  mkGen = mkOneHotGen;
endinstance
```

M. Synthesisable random generators

The mkGen function constructs a *synthesisable* pseudo-random generator using a linear congruential generator (LCG) [4], the same algorithm that is used in the GNU C library's implementation of the rand() function. One big attraction of an LCG is that it requires a small number of resources: in our case, 32 bits of state, an adder, and a constant multiplier. In the past, the multiplier may have been considered costly but it is not a problem on modern FPGAs with built-in DSP blocks. We use only the upper 16-bits of an LCG's state to obtain random numbers; this is because low-order bits are known to have a shorter period [4].

Typically BlueCheck will use several LCGs to test a given specification, each with a different seed. This is because we need to generate different random numbers *in parallel*. For example, in the stackSpec example, we need one generator to control the state machine (i.e. which property is being checked on any given clock-cycle), and another one to generate inputs to the push method. More generally, a single property may take several arguments, requiring a different generator for each

one. Currently, we make no attempt to share the output of a single LCG across several different generators, even though each generator may use only a small portion of an LCG's bits. Consequently, we waste some of the random bits available to us. This would be straightforward to resolve in future, if resource utilisation turns out to be problematic.

III. Case study

We have used BlueCheck in the development of BERI [2], a soft 64-bit multi-processor capable of running FreeBSD. One of the most challenging aspects of this implementation has been the cache-coherent shared memory subsystem, which comprises local direct-mapped L1 caches at each core and a single shared set-associative L2 cache. Below, we highlight the main lessons learned from using BlueCheck to test, and debug, the shared memory subsystem.

A. Interface

The hardware interface to the memory subsystem consists of just two methods: *put request* and *get response*. The complexity in the interface arises from the data types used to represent requests and responses: they are large structures containing many different fields. Some of these fields are only valid, or have different meanings, depending on the values of other fields. Consequently, these structures are quite difficult both to construct and to interpret. To abstract away from this complexity, we developed the following wrapper interface to the memory subsystem, known as a *memory client*.

```
interface MemoryClient;
  method Action load(Addr addr);
  method Action store(Data data, Addr addr);
  method ActionValue#(Data) getResponse;
  method Action setAddrMap(AddrMap map);
endinterface
```

Here, we have `load` and `store` methods to access 64-bit words in memory, and a `getResponse` method to obtain the result of a load. For presentation purposes, we have omitted several methods such as loads and stores of various widths, load-linked and store-conditionals, explicit cache flushes, memory barriers, and tagged memory operations. However, in practice we have indeed tested these additional methods using BlueCheck too.

As well as providing a human-readable interface, the memory client also solves an important problem: since the memory subsystem works on 64-bit addresses, it is extremely unlikely that random testing will ever generate the same address twice. To solve this, the memory client uses the `Addr` type for addresses, which consists of just four bits, and internally inserts these bits at arbitrary indexes in a blank 64-bit address. The actual indexes used can be set by calling the `setAddrMap` method, which takes a vector of four indexes, one for each `Addr` bit, as an argument. Note that this mapping is injective, i.e. each index must be different.

B. Specification

It is straightforward to define a golden memory client that performs no caching for equivalence testing against the memory subsystem. This leads to the following BlueCheck specification.

```
0  module [Specification] memSpec (Reset r);
1    // Implementation
2    MemoryClient mem <- mkMem(reset_by r);
3
4    // Golden model
5    MemoryClient gold <- mkMemGold(reset_by r);
6
7    // Before each test-sequence pick a random
8    // mapping from 4-bit addrs to 64-bit addrs
9    pre("setAddrMap", mem.setAddrMap);
10
11   equiv("load" , mem.load , gold.load);
12   equiv("store", mem.store, gold.store);
13   equiv("getResponse", mem.getResponse
14                      , gold.getResponse);
15 endmodule
```

On line 9, we use BlueCheck's `pre` function to pick a random address mapping at the beginning of each test-sequence. The idea here is that, while any one test-sequence will only use up to 16 different addresses, the actual 16 addresses used will vary *between* test-sequences. In order to generate an injective address mapping, we define a custom generator for the `AddrMap` type which ensures that the four random indexes are indeed different.

The main challenge that we encountered when implementing the above specification is that the DRAM component we connect to our memory subsystem does not initialise its contents on reset, and its capacity is too large to do an explicit intialisation. Consequently, it is not well-defined what the golden model should return when reading an uninitialised location. Fortunately, there are a number of solutions to this problem. The two simplest ones are:

(1) Modify the golden model's `getResponse` method so that it returns a special undefined `Data` value when an uninitialised location has been read. Then define equality on `Data` values to return true when one of the arguments is undefined.
(2) Considering that only 16 different addresses will ever be used in any single test-sequence, we can simply replace the DRAM component with a small on-chip memory and do an explicit intialisation. And with a little extra logic, we can also emulate a range of different DRAM latency and buffering parameters (something that our existing DRAM component did not do very well in simulation).

We chose option (2), mainly because (1) would result in many undefined loads, unless we refine the specification further.

C. Testing single-core access to memory

This test framework has proved extremely useful during a recent extensive refactoring of the memory subsystem by our implementation team, finding bugs almost on a daily basis. Most (but not all) of these bugs were are also exposable either by running our large suite of software unit tests, or by trying to boot FreeBSD. However, the main reason why our implementors prefer using BlueCheck is that it gives far simpler counter-examples. Perhaps the most extreme example of this occurred in a version of the memory subsystem capable of booting FreeBSD, but which led to the following counter-example using BlueCheck.

```
=== Depth 20, Test 82/10000 ===
setAddrMap(<13, 9, 3, 2>)
513: store(5,9)
516: load(8)
556: getResponse
557: load(9)
571: getResponse
571: Not equal: 0 v 5
```

It is quite surprising that an implementation containing a bug found by just five memory operations is capable of booting an OS involving millions of such operations! This particular bug arose from a cache prefetching feature. It is only exposable under certain conditions in which L2 cache lines are marked as invalid, and is hard to trigger once the L2 becomes populated. Typically, to diagnose such a bug, our implementation team will replay the test with cache debug messages enabled. This particular test leads to around ten debug messages. In contrast, one of our software unit tests that exposes the same bug leads to tens of thousands of debug messages. This highlights the value of searching for simple failures.

We test the memory subsystem both in simulation for rapid feedback and on FPGA for more thorough checking. Running at 90MHz on FPGA we check around 150K 30-element test-sequences per second. This compares to 350 sequences per second in simulation. An expected observation here is that benefit of on-FPGA testing increases with the amount of parallelism in the design under test: the benefit is two times greater for our parallel memory sub-system than for our earlier, largely-sequential stack example.

D. Testing multi-core access to memory

The above specification can be easily adapted to test multi-core memory accesses under the restriction that each core accesses different addresses. Although this does not test access to shared variables, it does invoke the coherency mechanism since that operates at the cache-line granularity and different addresses can map to the same cache line. Indeed, we have found bugs using this approach.

However, to properly test multi-core access, we must check that the memory subsystem satisfies a *memory consistency model* [5]. As such models are known to be challenging both to specify and to check against, we decided not to attempt an HDL-level specification. Instead, we used a more expressive software language to implement a tool *Axe* [6] capable of checking arbitrary memory traces against a range of consistency models, and connected it to a BlueCheck test bench via Bluespec's foreign function interface. Of course, this approach is not synthesisable, but it works well in simulation: we get the advantages of BlueCheck such as automatic generation of test-sequences and shrinking of counter-examples, along with the advantages of using a concise yet efficient software checker. More specifically, Axe uses the Yices constraint solver [7] to efficiently check the results of test-sequences consisting of load, store, load-linked, store-conditional, and memory barrier operations issued by multiple cores. At the time of writing, it supports checking against four consistency models: sequential consistency, total store order, partial store order, and relaxed memory order.

To illustrate, if we test our memory subsystem against Lamport's *sequential consistency* [8], then we obtain:

```
=== Depth 10, Test 5/10000 ===
setAddrMap(<15, 11, 8, 5>)
Core 0: MEM[3] == 0
Core 0: MEM[7] := 8
Core 1: MEM[3] := 9
Core 1: MEM[7] == 0
Core 0: MEM[3] == 0
Not sequentially consistent
```

BlueCheck finds what we believe is a minimal counter-example: there is no sequential interleaving of each core's operations that satisfies all the equalities shown. Compared to software litmus testing [9], which can also tell us that our memory subsystem is not sequentially consistent, the BlueCheck approach can do so while emitting far fewer cache debug messages, making it much easier to understand *why*. Using this approach, we have found that our memory subsystem satisfies a more relaxed model, total store order, on millions of random test sequences, giving us a high degree of confidence in its correctness. We have also found the approach very useful in exploring different cache coherency mechanisms, and rapidly determining the consistency models that they provide.

IV. HDL FEATURES

We have already seen (§II-C) two features of the Bluespec HDL that assist the implementation BlueCheck: (1) *implicit conditions* that automatically filter out ill-defined test-sequences such as popping from an empty stack; and (2) *atomic actions* that naturally enforce the requirement that both sides of an equivalence property must fire together or not at all. Without these features, users would have to carefully state the precise preconditions under which each property can fire. In this section, we discuss other features of Bluespec that assist the development of a generic test bench.

A. Output monads

A Bluespec module is an *output monad* [10] that implicitly collects *rules*: guarded atomic actions that are considered for execution on every cycle. Conveniently, the Bluespec authors made this monad generic enough to also collect user-defined data whose type is specified in square brackets after the `module` keyword. We use this feature to 'collect the specification' of the design-under-test, which is subsequently used to construct a test bench. To be more precise, let us consider the following module statement.

```
equiv("push", model.push, imp.push);
```

Executing this statement has the following effects:

1) It instantiates a psuedo-random generator for values of type t where t is the type of the argument to `push`.
2) It constructs an atomic action containing the two actions obtained by applying the function on each side of the equivalence to the output of the random generator. It then adds this atomic action to the specification.
3) It also adds the random generator to the specification. This is necessary because, to construct a test bench, we must be able access the random generators directly – to seed them and read their outputs – for saving and replaying counter-examples.

It is now apparent that when we say that a module 'collects the specification', it is in fact constructing pieces of the test

bench, just enough to let erase the types of the properties, and then collecting these simply-typed pieces. Had we tried to collect the properties directly, we would have required hetrogenous lists [11] (since each property can have an arbitrary but different type), and this approach leads to verbose types and error messages which can be confusing to the user.

B. Multi-cycle statements

In Bluespec, multi-cycle imperative-language statements can be placed in a `seq···endseq` block and automatically turned into a synthesisable finite state machine (in the style of [12]). We use this feature in order to concisely express the complex control flow present in BlueCheck's iterative-deepening and shrinking procedures. Constructing FSMs manually for these tasks would have been tricky and error prone.

C. First-class actions

Bluespec actions (sequences of RTL statements) are first-class values in the sense that, during static elaboration, they can be passed as arguments to functions, stored in data structures, and composed with other actions to form larger actions. This feature is heavily exploited in BlueCheck. For example, `equiv` statements yield actions that are collected in a data structure and later spliced into rules and `seq` blocks to form a test bench.

D. Sized FIFOs

Bluespec's arbitrary-sized FIFO module provides a very convenient way to iterate over a test-sequence. During test-sequence generation, the time of each property-invocation is inserted into a FIFO. The sequence can then be replayed by repeatedly deleting and re-inserting the head of the FIFO until all elements have been viewed. When shrinking is being performed, each FIFO element also contains a flag stating whether or not that property-invocation should be omitted.

E. Higher-order functions and type classes

BlueCheck properties are simply Bluespec functions whose arguments represent universally quantified variables. BlueCheck functions which accept properties as arguments, such as the `prop` function for registering a property, are therefore *higher-order*. To apply these user-specified properties to random data of the appropriate type, BlueCheck uses type classes [13]. Type classes may also be used to customise the psuedo-random generator for a particular type when the default generator for that type is not sufficient.

V. Related work

BlueCheck is heavily inspired by the QuickCheck library [1] for property-based testing in Haskell. Interestingly, one of the original motivations of QuickCheck was to test circuits written in Lava [14], an HDL embedded in Haskell. To our knowledge however, no attempt was made to do automatic test-sequence generation for stateful circuits in the style presented in this paper. The published experiences of testing stateful software using QuickCheck [15], another great source of inspiration to us, are certainly relevant to this topic.

The QuickCheck authors argue that high-level languages have a vital role to play in the verification of programs, regardless of the language used to implement those programs [16]. The reason is that high-level languages support rapid development of concise executable specifications against which implementations can be rigorously tested. A big limitation of the approach described in this paper is that Bluespec limits developers to writing synthesisable descriptions, and hence synthesisable specifications. The only way around this is to to use the foreign function interface which is somewhat clumsy. In this respect, a language like Lava has advantages because the full power of Haskell is conveniently available for writing high-level specifications.

A well-known issue with random testing is that it can be difficult to determine when testing has achieved sufficient coverage of the design under test. This is particularly problematic in our case as there is no source-code coverage tool currently available for Bluespec. As a result, an exhaustive testing strategy that can establish complete coverage of a predefined test space, such as that employed by SmallCheck [17], may be an attractive alternative to explore in future work.

Traditionally, verification support for mainstream HDLs has been provided in the form of language extensions and associated checking tools. A prime example of this is the *Property Specification Language* (PSL) [18] which has been successfully embedded into both VHDL and Verilog. PSL properties are comprised of boolean expressions, written in the host HDL, alongside new operators that allow *temporal* relationships between signals to be expressed. To illustrate, here is a PSL property, embedded in Verilog, asserting that a `req` signal must eventually be followed by an `ack` signal.

```
psl ReqThenAck: assert
  always (req -> eventually! ack) @posedge clk;
```

To check PSL properties, tool support for the PSL embedding is required, which is typically provided by EDA vendors in the form of advanced simulators or model-checkers.

More recently, the idea of a unified hardware description *and* verification language (HDVL) has emerged in the form of SystemVerilog [19]. It provides a powerful property language, similar to PSL [20], along with a number of other verification features. For example, random variables with arbitrary constraints may be declared, allowing concise definitions of test data generators satisfying complex invariants:

```
rand bit [3:0] a, b, c, d;
constraint sameSum { a+b == c+d; }
```

There is also support for expressing test-sequence generators using a notation inspired by context-free grammars:

```
randsequence (main)
  main      : repeat (10) pushOrPop;
  pushOrPop : push | pop;
  push      : { $display("push"); };
  pop       : { $display("pop"); };
endsequence
```

This simple generator displays a random 10-element sequence of `push` and `pop` strings. Another feature of SystemVerilog relevant to this paper is support for monitoring execution coverage, helping users to judge the completeness of testing.

In all, SystemVerilog offers a range of features that will no doubt improve productivity for hardware designers, especially

with regard to verification. However, the downside is that it requires a significant learning curve and substantial tool support. Indeed, none of the above-mentioned features are actually supported in FPGA vendor tools, and even the most advanced EDA tools, which can be expensive to obtain, only support simulation (not synthesis) of such features.

VI. Conclusions

Abstraction and reuse are two key tools for managing the complexity of modern hardware designs. For any developer intending to test components at the HDL level, this raises an important question: can common test bench features be usefully abstracted out and easily reused? In this paper, we have seen that the answer is a compelling 'yes': test-sequence generation, iterative-deepening, counter-example shrinking, equivalence checking, wedge detection, coverage monitoring, and error reporting can all be abstracted out in a fully synthesisable manner within a modern HDL. To use these abstractions, the developer simply provides a specification of correctness: a set of properties or equivalences that are expected to hold, which may be expressed within the HDL or using an external language. Since these abstractions can be used to test any design, we refer to them collectively as *a generic test bench*.

We have seen that our generic test bench can be easily applied in a range of different examples, and is capable of finding interesting bugs. The aim of a good test bench, though, is not just about finding bugs, but also helping to diagnose them. This is where our approach excels due to its ability to find *simple* failures – the main reason it has proven popular with our implementation team. Crucial to this success is our iterative-deepening strategy and shrinking procedure which together allow simple failures to be found quickly.

Our approach encourages the use of synthesisable specifications that can be rigorously tested on FPGA, allowing hundreds of times more tests to be explored per unit time compared to simulation. Crucial to this is the ability of the test bench to automatically transfer test-sequences to a host PC where they can be viewed or replayed in simulation. We have seen that such synthesisable specifications can be straightforward to write, even when the implementation under test is highly complex. We have also seen that in some cases it can be desirable to write specifications *externally* of the HDL using more expressive software languages. Although such cases are no longer synthesisable, BlueCheck's features can still be exploited in simulation.

Several features of the Bluespec HDL have aided the implementation of our generic test bench, including: atomic actions, implicit conditions, output monads, multi-cycle statements, first-class actions, higher-order functions, and type classes. Such features have made Bluespec ideal for exploring the ideas presented in this paper. An interesting avenue for future work is to see how similar ideas can be applied to other HDLs.

Much effort in the hardware verification community revolves around the development of language extensions and associated checkers. While these developments are all very important, they can sometimes require elaborate or hard-to-obtain tools. We hope this paper shows that there are also ways to support verification in a more lightweight manner, through a generic synthesisable test bench implemented simply as a library module in a modern HDL.

Acknowledgements

For many helpful suggestions, thanks to Alex Horsman, Alexandre Joannou, Theo Markettos, Peter Sewell, Robert Watson, Jon Woodruff, and the anonymous reviewers. This work was supported by DARPA/AFRL contracts FA8750-10-C-0237 (CTSRD) and FA8750-11-C-0249 (MRC2), and EPSRC grant EP/K008528/1 (REMS). The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

References

[1] K. Claessen, J. Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, in ICFP 2000, pp. 268–279.

[2] *BERI: Bluespec Extensible RISC Implementation*, http://beri-cpu.org/.

[3] J. Gannon, P. McMullin, R. Hamlet. *Data Abstraction, Implementation, Specification, and Testing*, ACM Transactions on Programming Languages and Systems, volume 3, number 3, pp. 211–223, 1981.

[4] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Third Edition, Addison-Wesley, 1997.

[5] S. Adve and K. Gharachorloo. *Shared Memory Consistency Models: A Tutorial*, Computer Journal, volume 29, number 12, pp. 66–76, 1996.

[6] M. Naylor and S. Moore. *A checker for shared memory consistency*, http://github.com/CTSRD-CHERI/axe.

[7] B. Dutertre, *Yices 2.2*, in CAV 2014, LNCS 8559, pp. 737–744.

[8] L. Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, volume 28, number 9, pp. 690–691, 1979.

[9] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, *Litmus: Running Tests Against Hardware*, in TACAS 2011, pp. 41–44.

[10] P. Wadler. *Monads for functional programming*, in Advanced Functional Programming, pp. 24–52, 1995.

[11] O. Kiselyov, R. Lämmel, and K. Schupke, *Strongly Typed Heterogeneous Collections*, Haskell Workshop 2004, pp. 96–107.

[12] I. Page and W. Luk. *Compiling Occam into Field-Programmable Gate Arrays*, in FPGAs, pp. 271–283, 1991.

[13] P. Wadler and S. Blott, *How to Make Ad-hoc Polymorphism Less Ad Hoc*, in POPL 1989, pp. 60–76.

[14] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. *Lava: hardware design in Haskell*, in ICFP 1998, pp. 174–184.

[15] K. Claessen and J. Hughes. *Testing monadic code with QuickCheck*, SIGPLAN Notices, volume 37, number 12, pp. 47–59, 2002.

[16] J. Hughes. *QuickCheck Testing for Fun and Profit*, in PADL 2007, LNCS 4354, pp. 1–32.

[17] C. Runciman, M. Naylor, and F. Lindblad. *Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values*, Haskell Symposium 2008, pp. 37–48.

[18] *IEEE Standard for Property Specification Language (PSL)*, IEEE Standard 1850-2005.

[19] *IEEE Standard for SystemVerilog: Unified Hardware Design, Specification, and Verification Language*, IEEE Standard 1800-2012.

[20] E. Cerny, S. Dudani, J. Havlicek, D. Korchemny. *The Power of Assertions in SystemVerilog*, Springer, 2010.