A Generic Synthesisable Test Bench

Matthew Naylor and Simon Moore

Email: *matthew.naylor@cl.cam.ac.uk*

Observation: HDL-level test benches are often written 'ad-hoc' with little or no code shared between them.

Can common test bench features be usefully abstracted out and easily reused?

BlueCheck:

- is a **generic** test bench that can be applied to any Bluespec module;
- is **parameterised** by correctness properties that are themselves expressed in Bluespec;
- automatically generates test-sequences and searches for **simple counter-examples**;

BlueCheck also:

- is **synthesisable**: failures found on FPGA are automatically transferred (over UART) to a host PC where they can be **viewed** or **replayed**;
- is a **pure Bluespec** library module (no pre-processors or language mods required);
- encourages the **formal specification** of hardware components through the reward of automatic testing.

Following QuickCheck

QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

ABSTRACT

QuickCheck is a tool which aids the Haskell programmer in formulating and testing properties of programs. Properties are described as Haskell functions, and can be automati-

monad are hard to test), and so testing can be done at a fine grain.

A testing tool must be able to determine whether a test is passed or failed; the human tester must supply an auto-

Testing: from a chore to an addiction!

Part I
BlueCheck by example

A stack in Bluespec

```
// Stack of 2^n items of type t

interface Stack#(type n, type t);
   method Action push(t x);
   method Action pop;
   method t      top;
   method Bool   isEmpty;
   method Action clear;
endinterface
```

A unit test in Bluespec

```
module unitTest ();
  Stack#(8, Bit#(4)) stk <- mkBRAMStack();

  Stmt testSequence =
    seq
      stk.push(1);
      stk.push(2);
      dynamicAssert(stk.top == 2, "Failed");
      stk.pop;
      dynamicAssert(stk.top == 1, "Failed");
    endseq;

  mkAutoFSM(testSequence);
endmodule
```

A specification in BlueCheck

```
import BlueCheck :: *;

module [Specification] stackSpec ();
  /* Implementation instance */
  Stack#(8, Bit#(4)) imp <- mkBRAMStack();

  /* Specification instance */
  Stack#(8, Bit#(4)) model <- mkRegStack();

  equiv("pop"     , model.pop     , imp.pop);
  equiv("push"    , model.push    , imp.push);
  equiv("isEmpty", model.isEmpty, imp.isEmpty);
  equiv("top"     , model.top     , imp.top);
endmodule
```

Instantiating BlueCheck

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

Running the test bench

Simulating `stackChecker` as top-level module:

```
push(13)
pop
push(6)
```

… 13 method calls elided …

```
pop
pop
top failed: 6 vs. 1
```

Handy Bluespec features

- **Implicit conditions** on methods act as a filter on ill-defined random test-sequences, such as popping from an empty stack.
- **Atomic actions** ensure that both sides of an equivalence property fire together or not at all.
- See Section IV of paper for more.

Resettable specifications

Enable a better testing strategy using **iterative deepening** and **shrinking**

```
module [Specification] stackSpecR (Reset r);
  /* Implementation instance */
  Stack#(8, Bit#(4)) imp <-
    mkBRAMStack(reset_by r);

  /* Specification instance */
  Stack#(8, Bit#(4)) model <-
    mkRegStack(reset_by r);

  /* Properties as before */
endmodule
```

Iterative-deepening & shrinking mode

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

Use iterative-deepening (ID)

Resettable specification

With iterative deepening enabled

```
=== Depth 2, Passed 5 tests ===
=== Depth 3, Passed 5 tests ===
=== Depth 4, Passed 5 tests ===
=== Depth 5, Passed 5 tests ===
=== Depth 6, Passed 5 tests ===
=== Depth 7, Test 4/5 ===
8: push(0)
9: push(9)
10: push(9)
17: push(4)
21: pop
22: pop
23: pop
24: top failed: 0 vs. 9
```

Tricky trade-off

Problem: how quickly to increase the depth?

- too fast: miss a small failure
- too slow: long time to find any failure

Tests-per-depth	Avg. time to first failure	Avg. size of first failure
10	1704	10
50	4468	7
100	7409	6
150	27388	5

With shrinking enabled

```
=== Depth 20, Test 1/10000 ===
6: push(9)
7: push(6)
8: push(13)
9: pop
10: pop
11: top failed: 9 vs. 6
Saved to 'CounterExample.bin'
Continue searching?
```

Shrinking finds small failures **quickly**

Replay-file produced in simulation or on FPGA

What about the `clear` method?

Problem: probability of generating stacks of more than two elements becomes rather low.

Solution: specify frequencies.

```
equivf(2, "pop"     , model.pop     , imp.pop);
equivf(4, "push"    , model.push    , imp.push);
equiv (   "isEmpty", model.isEmpty, imp.isEmpty);
equiv (   "top"     , model.top     , imp.top);
equiv (   "clear"   , model.clear   , imp.clear);
```

Relative frequencies (default is 1)

Parallel actions

Problem: BlueCheck assumes that action methods conflict with each other. But what about a stack allowing parallel push and pop?

Solution: specify parallel properties explicitly.

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

Algebraic properties

```
module [Specification] mkStackSpec ();
  Stack#(8, Bit#(4)) s1 <- mkBRAMStack();
  Stack#(8, Bit#(4)) s2 <- mkBRAMStack();

  function pushPop(s, x) =
    seq s.push(x); s.pop; endseq;

  function nop(s, x) = seq endseq;

  equiv("pushPop", pushPop(s1), nop(s2));
  equiv("push"   , s1.push    , s2.push);
  equiv("pop"    , s1.pop     , s2.pop);
  equiv("top"    , s1.top     , s2.top);
endmodule
```

Testing algebraic properties

```
=== Depth 20, Test 15/10000 ===
11: push(12)
22: push(2)
23: pushPop(14)
27: pop
28: top failed: 2 vs. 12
Saving to 'CounterExample.bin'
Continue searching?
```

Bug found without the need for a golden model

Other features

- Test generators for any data type can be **customised** when the default one doesn't suffice.
- Monitoring of **coverage** properties.
- **`pre`** and **`post`** properties.
- Wedge detection. This is an example of a **generic property**. (Future work: more generic properties.)

Part II

Application to CHERI’s memory subsystem

Single-core memory subsystem

- Has extremely complicated implementation but **simple specifications in BlueCheck**, both axiomatic and model-based.

- Model-based failures **easier to understand**.

- Tests load, store, capability load and store, cancellations, and cache management ops.

- **Hammers** memory subsystem and gives very concise counter-examples.

Heavily used for bug-fixing

In a CHERI build capable of booting FreeBSD:

```
=== 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 vs. 5
```

This leads to a **ten line** cache trace. Exposing the failure using our software unit test framework led to many **thousands of lines**.

Multi-core memory subsystem

- Some properties are **difficult to express in Bluespec**, which is largely limited to synthesisable descriptions.

- Prime example: checking shared memory consistency (cache coherency, barriers, atomics).

- We developed a consistency checker (Axe) in Haskell, and connected it to Bluespec via its **foreign function interface**.

CHERI's memory model

BlueCheck tells us it is TSO, and reports this counter-example to sequential consistency:

```
=== 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
```

Compared to software litmus testing, failing cache traces are **much** smaller (tens of lines vs. millions).

Conclusion

Can common test bench features be usefully abstracted out and easily reused?

Yes: test sequence generation, equivalence checking, iterative-deepening, shrinking, FPGA to PC error reporting, coverage monitoring, wedge detection, all achieved generically.

https://github.com/CTSRD-CHERI/bluecheck