Verifying Verification

Using Open Source to roll your own

David Hewson
Why mutation testing

● The test plan
  ● Defines what you need to exercise
  ● Defines what you should be checking

● Functional coverage
  ● Measures what you have exercised

● Assertion coverage
  ● Can indicate that your checks have been exercised

● But how to find bugs in the test benches?
  ● Inject errors into the design and see if they’re caught
Why consider rolling your own?

● **Commodity hardware**
  ● Fast multi-core processors
  ● Desktop motherboards
  ● Low cost memory
  ● No storage
  ● Minimal packaging
  ● Less than £50 per core

● **Open Source software**
  ● Verilator – a Verilog simulator
  ● Grid engine – batch system
  ● Verilog Perl – Verilog language parser
What is Verilog Perl

- Part of the VeriPool software
  - Publicly licensed open source software
  - Written by Wilson Snyder <wsnyder@wsnyder.org>

- A building point for Verilog support in the Perl language
  - Verilog::Language
    - Provides support for the Verilog language keywords
  - Verilog::Preproc
    - Preprocessor for Verilog language
  - Verilog::Parser
    - A parser for Verilog which invokes call-backs for language tokens
    - Supports all SystemVerilog 2012 keywords
Extending the parser

● **We extended the parser to recognise expressions**
  ● Just a matter of inheriting from the class and implementing the appropriate call-backs
  
  ● For us it was sufficient just to recognise expressions for ‘wire’ and ‘assign’ statements – it would be easy to extend for all others

● **Build an abstract syntax tree**
  ● Unary Operators  - eg |a, &a, ~a
  ● Binary Operators - eg a|b, a&b, a==b, a||b
  ● Concatenations  - eg {a,b}
  ● McCarthy expressions - eg a ? b : c
  ● Symbols        - eg a, b, c
Generation of mutations

● **Define mutations for each type of AST Node**
  ● Unary, Binary operands
    ● Replace with operand of same geometry
  ● Binary operands
    ● Remove LHS or RHS
    ● Re-order LHS and RHS
  ● Concatenations
    ● Re-order the elements
  ● Symbols
    ● Replace with inverse

● **Build a list of code changes**
  ● Iterate over the AST applying the change
  ● Output the modified code as a “diff” to apply
Example

module example (clk, ra, rb, rc, rd, re, res)
  input clk, ra, rb, rc, rd;

  wire res_N = {ra, rb + rc, rd | re };

  always @(posedge Clk)
    res <= res_N;
endmodule

example.v

wire res_N = 0;
wire res_N = {ra, "rb + ("rc), "rd | re };
wire res_N = {ra, rb + rc, rd & re };
wire res_N = {"ra, rc, rd | re };
wire res_N = {"ra, "rc, re };
wire res_N = {ra, "rb , rd };
wire res_N = {ra, rb + ("rc), re };
wire res_N = {ra, rb - rc, rd | re };
wire res_N = {"ra, "rc, rd | ("re )};
wire res_N = {"ra, "rc, "rd };
wire res_N = {"ra, rb , "rd | ("re )};
wire res_N = {ra, rc, rd | ("re )};
wire res_N = {ra, "rc, "rd };
wire res_N = {ra, "rb - rc, rd ^ ("re )};
wire res_N = {"ra, "rb + rc, "rd | ("re )};
Running the tests

● **For each mutation we would**
  ● Created a mutated version of the source file
    ● Remove the required lines and insert the modified ones
  
  ● Compile the design with the mutated code
    ● For Verilator this was a full re-compile
    ● For commercial simulators, the mutated code was built into a library
  
  ● Execute the test suite until the first failure
    ● Use seeds from our best coverage run as well as random seeds
    ● Prioritise tests by where the mutation is applied
  
  ● Re-run the failing test against the original design
    ● We may have stumbled across a real bug!
Each mutation test is completely independent
- Each one involves a potentially large number of tests to execute
- Hence we have a vast number of concurrent jobs we can run
  - Verilator has no licensing limitations

We had to adapt our build/run scripts
- To include compiling the mutated source file
- We were using Cadence emanager for coverage runs
  - Develop scripts for submitting a test sequence to batch system

The Grid Engine manages the resource allocation
- We run mutation testing at a lower priority to soak up idle time
- We cancel jobs for a mutation after the first failure is detected
Challenges we faced

- We needed to ensure that a positive error was detected
  - Some mutations could result in slow performance or never complete

- Correlation between mutation and missing/failing check
  - It was quite time consuming to determine what the effect should be
  - This involved both RTL designer and verification engineer
  - But this is similar to understanding expression coverage

- Some Verilog code would yield mutations with no discernable effect
  - Could indicate dead code was present
  - Could have formal equivalence checked to filter these cases

- What if you don’t have 100% expression coverage
  - Anything that wasn’t covered is unlikely to be checked
  - Could potentially interrogate coverage data to exclude them
Conclusions

● As a result we did find bugs in the design
  ● It was worth the couple of months taken to develop
● Mutation testing is just one way of improving quality
  ● Treat the verification components as a software development project
    ● Unit tests for verification components
    ● Measure code coverage
    ● Use continuous integration
● Parser and mutation generator have been donated to TVS

● You can never have too much compute power