AUTOMATED BEHAVIORAL TESTING OF VHDL COMPONENTS

Peter Walsh
Department of Computing Science, Malaspina University-College
900 Fifth Street, Nanaimo, B.C., V9R 5S5 Canada

Daniel Hoffman
Department of Computer Science, University of Victoria,
P.O. Box 3055, Victoria, B.C., V8W 3P6 Canada

ABSTRACT

Design reuse is essential for dealing with complex integrated circuits and printed circuit boards. The use of a hardware description language such as VHDL is becoming a requirement for this kind of reuse. The current practice is to model the design component in VHDL and to test the design's behavior using a testbench, which applyies test cases and reports deviations of actual behavior from expected behavior. In this paper we present VHDLGEN, a prototype method and tool for algorithmic testing of VHDL designs, adapted from a successful approach to testing software components. Testing with VHDLGEN involves a special interface to the component under test, a scripting language, and a driver generator. VHDLGEN is oriented towards highly automated testing, where the driver generates the inputs, runs the tests, and checks the outputs for correctness. We present evidence that the VHDLGEN approach is feasible. Our experience with algorithmic testing of software components suggests that VHDLGEN will scale up to more complex designs and will be adaptable to designs in object-oriented HDLs as they become available.

1. INTRODUCTION

From a testing viewpoint, hardware designs in VHDL are similar to conventional software programs. When testing a VHDL design, the focus is on errors in the VHDL, not in the fabrication process to come. When testing software, the focus is on errors in the code, not in the compiler. In both cases the main source of problems is coding errors. This similarity suggests that techniques that have been successfully used to test software may also be effective when applied to testing hardware designs in VHDL. Because VHDL designs are for ICs or cells, it makes sense to focus on techniques used for testing software components, as opposed to large software systems.

For software components, thorough testing requires a large number of test cases; thousands of test cases are not uncommon. These test cases must be run many times: after each version due to bug fixes or enhancements, and after each change in the environment including the compiler and support libraries. With many tests that must each be run many times, it is critical that the cost of each test run be low. Ideally there would be no cost except for machine time. Only highly automated testing can achieve these low costs. In particular, the checking of the correctness of the outputs must be automated.

Many programmers test their code interactively, using a debugger or a driver program that prompts them for inputs and displays outputs. The test inputs are entered with the keyboard and mouse and the outputs are checked visually. This approach is good for debugging, where the next test input is not known until the results of the current input have been seen. However, interactive testing is expensive and unreliable when many test cases must be repeated many times.

With algorithmic testing, the test inputs are generated by a program and the outputs are checked by a program. Algorithmic testing provides cheap and reliable regression testing. The cost of developing the test programs must be considered, but once developed, the cost of repeating the tests is negligible.

Modern capture/playback tools support a hybrid testing approach. Inputs are entered by hand but "captured" for later playback. In some cases, outputs may also be recorded. This approach does allow tests to be repeated cheaply and reliably. However, capture/playback tests are expensive to develop if many tests are required. Further, the scripts are very sensitive to change and are thus expensive to maintain [7].

PGMGEN is a test driver generator originally developed to test C modules [2]. Other versions have been developed and used to test Fortran, assembler, and C++ [1, 6]. ClassBench is a framework specifically designed for testing C++ class libraries [3].

VHDLGEN was adapted from the PGMGEN tool to test VHDL designs. With the current efforts to add OO features to VHDL [8], we believe that the ClassBench approach may soon be applicable to VHDL.

2. BEHAVIORAL TESTING IN VHDL

The block-diagram characteristics of an electronic component are defined in VHDL *entity declarations*. VHDL *architecture declarations* define component behaviour. The goal of behavioural testing is to establish that a component exhibits certain required behaviour. To this end, test cases are applied to the component and the observed behaviour is checked against the required behaviour.

Debuggers and interactive drivers are used to interactively test the behaviour of VHDL components. For non-interactive testing, the current practice is to develop a testing sub-system known as a testbench. A testbench consists of an implementation of the component under test (CUT), a driver component, and fully functional implementations of any other components that interact with the CUT. The testbench driver is used to exercise the sub-system. Its job is to apply test cases to the CUT and report any deviations in actual behaviour from expected behaviour.

From the tester's perspective, the ideal is to completely isolate the CUT and exercise it using the driver. The testbench approach described above is less than ideal since it may contain components other than the driver and the CUT. The presence of these extra components in the testbench is undesirable since they may adversely effect test controllability and observability.

Two types of driver are common: one is algorithmic and the other reads test vectors from an external file [4]. Drivers are usually hand-coded even though a significant proportion of the code is boiler-plate in nature. Drivers that read test vectors from an external file are expensive to maintain. Often, when a design component evolves and is reused, its testbench must be re-developed from scratch.

3. VHDLGEN

VHDLGEN is a CAD tool that can be used to automatically generate a driver for a VHDL testbench. In contrast to software modules, VHDL components can be active. Components may interact within a testbench outside the control of the driver. As alluded to earlier, this can cause controllability and observability problems.

We address this problem by accessing the CUT through a testing interface. The testing interface makes the CUT reactive. That is, the CUT only responds to stimuli from the driver.

To illustrate the use of VHDLGEN, consider the design of simple Logical And (`LAnd`) gate. Let `LAnd` have two binary inputs, `A` and `B`, and one binary output `C`. In terms of behaviour, `C = A.B` and, the value of `C` is valid 10ns after a change in value to either `A` or `B`. The entity declaration for `LAnd` is:

```
entity LAnd;
   port (A,B: in bit; C: out bit);
end LAnd;
```

The testing interface consists of the following VHDL procedures and functions:

```
procedure LAInit is
begin
   -- empty
end;

procedure LASetInputs(x,y: in bit) is
begin
   A <= x;
   B <= y;
   wait for Td; -- time delay
end;

function LAGetOutput return bit is
   variable x: bit;
begin
   out_capture(x); -- capture value
                   -- of signal C
   return (x);
end;
```

`LAInit` is an initialization procedure and it is included to facilitate design for testability. The other subprograms allow us to set inputs and capture outputs.

The syntax of a test-case in VHDLGEN is *tcase(trace, actval, expval)*. A *trace* is any sequence of calls to the CUT through the testing interface. The *actval* is an expression that is evaluated after the trace. It's value is taken as the "actual value" of the trace. *expval* is the value that *actval* is expected to have. Four test cases are necessary to exhaustively test the behaviour of `LAnd`. They are:

```
tcase(LAInit;                   -- trace
      LASetInputs('0','0'),
      LAGetOutput,              -- actual value
      '0')                      -- expected value
tcase(LAInit;
      LASetInputs('0','1'),
      LAGetOutput,
      '0')
tcase(LAInit;
      LASetInputs('1','0'),
      LAGetOutput,
      '0')
tcase(LAInit;
      LASetInputs('1','1'),
      LAGetOutput,
      '1')
```

In general, a test case takes the form *tcase(*$c_1; c_2; ..; c_N$*, actval, expval)*. VHDLGEN generates VHDL code to

- invoke $c_1; c_2; ..; c_N$
- if $actval \neq expval$ then print a message
- update summary statistics

4. EXPERIENCE

Our case study is the design of a controller (`TLight`) for a traffic light at the intersection of a highway and a farm-road. It is a variation of the controller given in [5]. The behaviour of the controller is as follows. The highway lights remain green as long as there are no cars on the farm-road. When the highway lights have been green for a minimum of ten seconds, a car on the farm-road will cause the highway lights to go yellow for three seconds and then red. Then the farm-road lights will turn green and remain green for a minimum of three seconds. The presence of other cars on the farm-road beyond the three seconds will enable the farm-road lights to remain green for a maximum of ten seconds. The farm-road lights will then go yellow for three seconds and then red.

The controller has four states, highway green (`HG`), highway yellow (`HY`), farm-road green (`FG`) and farm-road yellow (`FG`). In state `HG`, the highway lights are green and the farm-road lights are red. In state `HY`, the highway lights are yellow and the farm-road lights are red. In state `FG`, the farm-road lights are green and the highway lights are red. In state `FY`, the farm-road lights are yellow and the highway

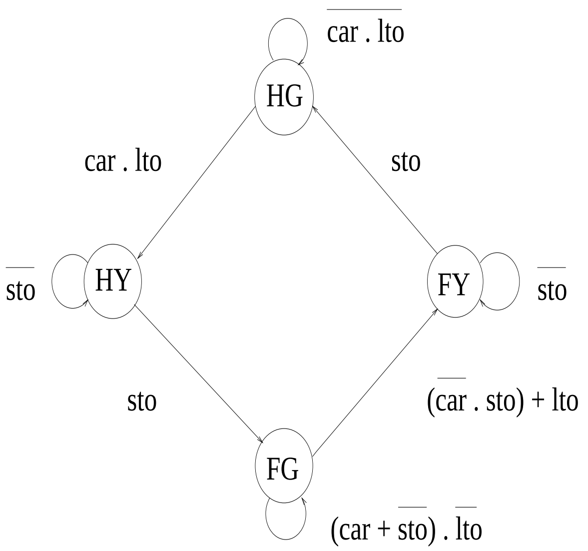

Figure 1. Traffic Light Controller FSM

lights are red. The controller has five control signals: `clear`, `car` , `sto`, `lto` and `clk`. The control signal `clear` is asynchronous. All other control signals are synchronous. When `clear` is asserted, the controller resets to state `HG`. When `car` is asserted, a car is present on the farm-road. When `lto` is asserted, a long-delay timer has reached the ten second mark. When `sto` is asserted, a short-delay timer has reached the three second mark. State transitions are shown in Figure 1. The entity declaration for `TLight` is:

```
entity TLight is
   port (clear, car, sto, lto, clk: in bit;
   hwred, hwyellow, hwgreen, farmred,
   farmyellow, farmgreen: out bit);
end;
```

There are five single-bit inputs and six single-bit outputs. The five inputs correspond to the control signals described earlier. Three of the outputs correspond to the red, green and yellow lights visible from the highway. The other three correspond to the red, green and yellow lights visible from the farm-road.

The following VHDL definitions are used in the driver.

```
constant Td:time:= 20 ns;
constant Id:time:= 5 ns;
constant HGLIGHTS: bit_vector := "001100";
constant HYLIGHTS: bit_vector := "010100";
constant FGLIGHTS: bit_vector := "100001";
constant FYLIGHTS: bit_vector := "100010";
constant NumOutputs: integer := 6;
constant NmInp: integer := 3;
constant MaxInpVal: integer := 7;
-- 2 to the power of NmInp less 1

type LightState is (HG, HY, FG, FY);

variable s: LightState;
variable i, choice: integer;
variable test_vector: bit_vector (0 to NmInp-1);
alias t_car: bit is test_vector(0);
alias t_sto: bit is test_vector(1);
alias t_lto: bit is test_vector(2);
```

The constant `Td` is the time the clock is asserted. The constant `Id` is the minimum time that `clear` must be asserted to reset the controller. The constants `HGLIGHTS` through `FYLIGHTS` are output bit patterns corresponding to valid traffic light settings. The variable `test_vector` is used to accumulate test values for the `car`, `sto` and `lto` signals. It ranges in value from zero to `MaxInpVal`.

The testing interface for `TLight` consists of the following procedures and functions:

```
procedure TLInit  is
begin
   clear<='1', '0' after Id;
   wait until clear'event and clear='0';
end;

procedure TLSetInputs
          (x: in bit_vector(0 to NmInp-1)) is
begin
   car<=x(0);
   sto<=x(1);
   lto<=x(2);
   clk<='1', '0' after Td;
   wait until clk'event and clk='0';
end;

function TLGetOutputs return bit_vector is
   variable x: bit_vector (0 to NumOutputs-1);
begin
  out_capture(x); -- capture output bit vector
  return (x);
end;

procedure TLLoadState(n: in LightState) is
begin
   case n is
      when HG => TLInit;
      when HY =>
         TLInit;
         TLSetInputs("101");
      when FG =>
         TLInit;
         TLSetInputs("101");
         TLSetInputs("010");
     when FY =>
         TLInit;
         TLSetInputs("101");
         TLSetInputs("010");
         TLSetInputs("001");
  end case;
end;
```

`TLinit` asserts the control signal `clear`. `TLSetInputs` allows for the selective assertion of the control signals `car`, `sto` and `lto`. `TLGetOutputs` returns controller output. `TLLoadState` sets the state of the controller. It is included to aid testability.

We have developed test cases to exhaustively test the behaviour of the controller. The following is an extract from our test script:

```
tcase(TLLoadState(HG), TLGetOutputs, HGLIGHTS)
tcase(TLLoadState(HY), TLGetOutputs, HYLIGHTS)
tcase(TLLoadState(FG), TLGetOutputs, FGLIGHTS)
tcase(TLLoadState(FY), TLGetOutputs, FYLIGHTS)

for s in HG to FY loop
   for i in 0 to MaxInpVal loop
      test_vector := conv_bit_vector(i,NmInp);
      case s is
         when HG =>
            if not ((t_car='1') and
                   (t_lto='1')) then
               tcase(TLLoadState(s);
                     TLSetInputs(test_vector),
                     TLGetOutputs,
                     HGLIGHTS)
            else
               tcase(TLLoadState(s);
                     TLSetInputs(test_vector),
                     TLGetOutputs,
                     HYLIGHTS)
            end if;
         when HY =>
            ...
         when FG =>
            ...
         when FY =>
            ...
      end  case;
   end loop;
end loop;
...
```

The first four test cases check for correct output when the controller is in one of its four states. For a particular state, the inner `for` loop applies all possible input values for the control signals `car`, `sto` and `lto` to the CUT and, checks for correct state-transition and output. The outer `loop` repeats the process for all controller states.

Notice how the testing interface provides a view of the CUT that allows the test-engineer to concentrate on behavioural verification. The test-engineer is not distracted by the CUT's interaction with other components. This is taken care of within the testing interface. This separation of concerns increases test controllability and observability and, facilitates the construction of test code that is more maintainable.

Exhaustive testing is not feasible for systems with a large number of states and a large number of inputs. For large systems, we select our tests-cases in a manner analogous to selecting test cases for software modules [2].

5. SUMMARY

We have described several approaches to testing VHDL designs. While interactive testing and keystroke capture/playback are widely used, algorithmic testing has significant advantages when many test cases must be run many times. As VHDL designs increase in complexity, the advantages of algorithmic testing will increase as well.

We have presented VHDLGEN, a prototype method and tool for algorithmic testing of VHDL design, adapted from a successful approach to testing software components. Testing with VHDLGEN involves a special interface to the component under test, a scripting language, and a driver generator. VHDLGEN is oriented towards highly automated testing with tests that are inexpensive to develop, execute, and maintain.

We have presented evidence that the VHDLGEN approach is feasible. Our experience with algorithmic testing of software components suggests that VHDLGEN will scale up to more complex designs and will be adaptable to test designs in object-oriented HDLs as they become available.

REFERENCES

[1] P. Brown and D.M. Hoffman. Application of module regression testing at TRIUMF. *Nuclear Instruments and Methods in Physics Research*, A293(1,2):377–381, August 1990.

[2] D.M. Hoffman. Hardware testing and software ICs. In *Proc. Pacific Northwest Software Quality Conf.*, pages 234–244, September 1989.

[3] D.M. Hoffman and P.A. Strooper. The testgraphs methodology: Automated testing of collection classes. *Journal of Object-Oriented Programming*, pages 35–41, Nov/Dec 1995.

[4] S. Mazor and P. Langstraat. *A Guide to VHDL*. Kluwer Academic Publishers, 1993.

[5] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[6] G.C. Murphy, P. Townsend, and P.S. Wong. Experiences with cluster and class testing. *Commun. ACM*, 37(9), September 1994.

[7] A. Polk. Testing GUI applications without using the GUI. In *Proc. 8th Intl. Software Quality Week*. Software Research Inc., May 1995.

[8] P. Swamy, A. Molin, and B. Covnot. OO-VHDL object-oriented extensions to VHDL. *IEEE Computer*, pages 17–26, October 1995.