Applying Test-Driven Development Methods to Design Verification Software

Doug Gibson (doug.gibson@hp.com), Mike Kontz (michael.kontz@hp.com)
Hewlett-Packard Company
Rationale

- Hardware DV Engineers are usually not trained software engineers.
- DV Software is not the product - it’s only a means to an end.
- As small projects become big projects, development needs to become more disciplined.
- Reusable DV software should be validated in a usage-independent manner.
How do we engineer DV software?

• In an effort to improve the quality of our DV software, we invested in some software development process training
• Better OO design principles
• Exploration of agile best-practices
• Test-driven development – unit test

Build a testbench for your DV software
What is Test-driven Development?

Consider the development of a software “feature” – either a new object or a new method of an existing object:

**Conventional Development**
- Design the feature
- Write the code that implements the feature
- Run the code within the RTL testbench if available
- Debug later if defects should arise

**Test-driven Development**
- Design the feature
- Write unit-tests that stimulate the feature in a standalone environment
- These tests should initially “fail”
- Write the code that implements the feature
- Debug tests that fail as necessary
- Continue until all tests “pass”
- Stop coding
What is Unit Testing?

- Test in isolation that a unit of code behaves correctly under different stimuli

- Code behaviors tested can include:
  - method return values
  - method actions (eg state update, external method calls)
  - error detection and messages
  - no errors if good stimulus
  - object final state
  - higher level objectives (eg counting events, output randomness)

- Example:
  - **Input**: \( \text{input} = 5 \text{ divided by } 0 \)
  - **Output**: \( \text{output} = \text{"ERROR: Cannot divide by zero"} \)
Testsuites and Testcases

- A testsuite is a set of testcases.

- A testcase creates a set of objects, performs a set of operations on those objects, and checks for proper response from the objects.

- Each testcase is isolated from all others. Execution order shouldn’t matter.
Some more terminology

- **Unit/Feature Under Test (UUT/FUT)** - component/feature being tested
- **Mock-Up Units** - “mock” versions of the required components to run and test the UUT
- **Assertions** - helper methods to check the desired UUT behavior
  - `expect <expr> to be <expr>`
  - `expect_dut_error <string>`
  - `expect_no_more_dut_errors`
  - `wait_for_expected_event <event> <within cycles>`
CppUnit  (See http://cppunit.sourceforge.net)

• Originally developed our methodologies around the open-source CppUnit framework

• Focused on checker/scoreboard development
  • Most expensive components in the testbench
  • Productivity and schedule are critical
  • Bugs have a huge impact on product schedule and quality

```cpp
class ComplexNumberTest : public CppUnit::TestCase {
public:
    ComplexNumberTest( std::string name ) : CppUnit::TestCase( name ) {} 

    void runTest() {
        CPPUNIT_ASSERT( Complex (10, 1) == Complex (10, 1) );
        CPPUNIT_ASSERT( !(Complex (1, 1) == Complex (2, 2)) );
    }
};
```
So what about UVM-e?

• You can do the same thing in UVM-e, but...

• Some UVM-e objects cannot be created/destroyed, namely UVM-e units. How do I create a set of testcases?!!

Enter the UVM e-Unit Testing Framework

• A standard mocking test framework – testcases contained in testsuites
• Each testcase instantiates a new testsuite and mockup unit structure
• Tests can assert result values and “expect” dut_error messages
• Emission of Specman-e events can be tested for
• A standard framework allows for testsuites to be distributed with VIP
About our Examples

UVM Reference Flow from Accellera

Provides a readily available working example of UVM-e code.

UVC’s are reasonably simple to understand.
So what does a test look like?

```
add testcase calc_parity to uart_frame with scenario {
    var fut_result := p_testsuite_uart_frame.uart_frame_s.calc_parity(
        p_testsuite_uart_frame.input_payload,
        p_testsuite_uart_frame.input_parity_type);

    eu_expect fut_result to be 1;
}
```
How about a TestSuite?

Create a TestSuite

Add a UUT

Run some setup

Add add’l mock objects

Complete construction

```plaintext
unit uart_env_testsuite like eu_testsuite {
    keep
    kind const uart_env;

    // The struct under test
    uart_env_u : TRUE 'has_tx TRUE 'has_rx uart_env_u is instance;

    unit_tests_setup() is also {
        sig_sec_cdma_tx_data$ = 1;
        sig_sec_cdma_rx_data$ = 1;
    };

dummy_p_sync : UART_ENV_MOCKUP 'eu_kind uart_env uart_sync is instance;
keep soft uart_env.u.p_sync == dummy_p_sync;

    sig_sec_cdma_tx_data : inout simple_port of bit is instance;
    sig_sec_cdma_ctsb : inout simple_port of bit is instance;
    sig_sec_cdma_rx_data : inout simple_port of bit is instance;
    sig_sec_cdma_rfrfb : inout simple_port of bit is instance;

    connect_ports() is also {
        uart_env_u.tx_agent.ssmpr.sig_sec_cdma_ctsb.disconnect();
        do_bind(uart_env_u.tx_agent.ssmpr.sig_sec_cdma_ctsb, sig_sec_cdma_ctsb);
        uart_env_u.tx_agent.ssmpr.sig_sec_cdma_tx_data.disconnect();
        do_bind(uart_env_u.tx_agent.ssmpr.sig_sec_cdma_tx_data, sig_sec_cdma_tx_data);
        uart_env_u.rx_agent.ssmpr.sig_sec_cdma_rfrfb.disconnect();
        do_bind(uart_env_u.rx_agent.ssmpr.sig_sec_cdma_rfrfb, sig_sec_cdma_rfrfb);
        uart_env_u.rx_agent.ssmpr.sig_sec_cdma_rx_data.disconnect();
        do_bind(uart_env_u.rx_agent.ssmpr.sig_sec_cdma_rx_data, sig_sec_cdma_rx_data);
    };
};
```
Monitor Testing

- Checking and maintaining the quality of your BFM is relatively simple.
- Doing the same for your monitor and its packet level checking is not. This checker is critical to quality verification, and yet may NEVER report any errors.
Packet-level protocol checker testing

- Loop over permutations
- Configure the UUT
- Make an input frame packet
- Create raw data for monitor input
- Send a legal frame
- There should be no errors!

```
add testcase check_receive_data to uart_env with scenario {
  var frame : uart_frame_s;
  var out_data : list of bit;
  var bfm := p_testsuite_uart_env.uart_env_u.tx_agent.as_a(ACTIVE TX uart_agent_u).bfm.as_a(TX uart_bfm_u);
  var monitor := p_testsuite_uart_env.uart_env_u.rx_agent.as_a(has_monitor RX uart_agent_u).monitor;
  var config := p_testsuite_uart_env.uart_env_u.config;
  for each (parity) in all values(uart_frame_parity_t) {
    for each (stopbit) in all values(uart_frame_stopbit_t) {
      for each (databit) in all values(uart_frame_databit_t) {
        config.parity_type = parity;
        config.stopbit_type = stopbit;
        config.databit_type = databit;
        gen frame keeping {
          it.parity_type == parity;
          it.stopbit_type == stopbit;
          it.databit_type == databit;
          it.legal_frame == TRUE;
        };
        var data : list of bit = pack(packing.low,frame);
        var parity_loc : uint = (1 + databit.as_a(int));
        var stop_loc : uint = parity_loc + ((parity!=NONE)?1:0);
        var orig_data := data.copy();

        // Check with no error
        monitor.check_receive_data(data);
        eu_expect_no_more_dut_errors;
    }
  }
}
```
Packet-level protocol checker testing

Start with legal data and flip the parity bit

Validate that a dut_error is emitted by the checker

And ensure that no other dut_errors appear

```cpp
// Corrupt the parity bit
if (parity!=NONE) {
    data = orig_data.copy();
    data[parity_loc] = ~data[parity_loc];
    monitor.check_receive_data(data);
    eu_expect_dut_error "Frame has bad parity";
    eu_expect_no_more_dut_errors;
}

// Corrupt the stop bit
data = orig_data.copy();
data[stop_loc] = ~data[stop_loc];
monitor.check_receive_data(data);
    eu_expect_dut_error "Frame stop bit is not correct";
    eu_expect_no_more_dut_errors;
```
We found bugs in the reference code!

- The `uart_monitor` wasn’t checking anything!
  - The `do_check` configuration flag was getting set to FALSE, likely due to a refactor. This disabled all `uart_frame` checking.

- The `with_parity` flag in the packet was not set correctly by the monitor!
  - When the packet contained parity bits, they were not checked for correctness
  - Probably due to a performance enhancement
Let’s test the Scoreboard

• Like the packet-level checker, the Scoreboard is a critical piece of the DV strategy.
• How do you know it’s working?
Scoreboard Testing

Generate an input
uart_frame

Generate an output apb
bus transaction

Emulate the input from
the uart monitor

Emulate the input from
the apb bus monitor

This sequence should
cause no dut_error

```java
import testsuite_uart_ctrl_scbd_monitor.e;

// Test uart_frame->apb path with correct payload
add testcase uart_input to uart_ctrl_scbd with scenario {
    var input_apb_trans : apb_trans_s;
    var uart_frame : uart_frame_s;
    gen uart_frame;
    gen input_apb_trans keeping {
        .addr == UART_RX_FIFO;
        .direction == READ;
        .data == pack(NULL,uart_frame.payload);
    };
    p_testsuite_uart_ctrl_scbd.ports_bundle.mock_uart_frame.add$.write(uart_frame);
    p_testsuite_uart_ctrl_scbd.ports_bundle.mock_apb_trans_match$.write(input_apb_trans);
    eu_expect_no_more_dut_errors;
}
```
How about a Scoreboard mismatch?

```c
// Test uart_frame->apb path with incorrect payload. Expect error
add testcase uart_input_bad_payload to uart_ctrl_scbd with scenario {
  var input_apb_trans : apb_trans_s;
  var uart_frame : uart_frame_s;
  gen uart_frame;
  gen input_apb_trans keeping {
    .addr == UART_RX_FIFO;
    .direction == READ;
    .data == (1 ^ pack(NULL, uart_frame.payload));
  };
  p_testsuite_uart_ctrl_scbd.ports_bundle.mock_uart_frame_add$.write(uart_frame);
  p_testsuite_uart_ctrl_scbd.ports_bundle.mock_apb_trans_match$.write(input_apb_trans);
  eu_expect_dut_error "Mismatch";
  eu_expect_no_more_dut_errors;
};
```
Using the tool

- Integrating a tool like AMIQ’s DVT-Eclipse can improve productivity
- How can we add unit testing to the development flow?
- What about continuous integration?
Unit Tests vs Turnon and Demo tests

- It is common for VIP to include standalone demo tests and undergo turnon testing. How does Unit Test differ from these kinds of tests?

<table>
<thead>
<tr>
<th></th>
<th>Turnon</th>
<th>Demo</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Testbench</strong></td>
<td>Full topology with RTL</td>
<td>Standalone + Demo Objects</td>
<td>Standalone + Mock-Ups</td>
</tr>
<tr>
<td><strong>Configurations</strong></td>
<td>1 configuration per topology</td>
<td>1 configuration per example</td>
<td>Iterates through all configurations</td>
</tr>
<tr>
<td><strong>Features tested</strong></td>
<td>Core functionality</td>
<td>Basic operation</td>
<td>• All features</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Corner cases</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Error reporting</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Event signaling</td>
</tr>
</tbody>
</table>
Unit Test Benefits

• Real Unit Test benefits experienced by our lab:
  • Verification IP development starts earlier
    − No RTL needed to begin full features and thorough testing.
  • Shorter development time
    − Testing is done coincident or immediately after development while the design is fresh in our heads.
    − Development + Testing cycle is continuous as unit test results are quick. Less need to multitask.
    − Less debugging in larger, complex environments. Issues found in small unit tests.
  • Fewer verification holes
    − Error detection and messaging has been unit tested. All configurations covered.
  • RTL turnon really is just RTL turnon
    − Verif components are already tested and working. Validation portion of project stays focused on RTL issues and testing.
  • Faster and better quality fixes for verif components
    − Reproduce bug with unit test. Rerun unit test library to make sure fix doesn’t introduce new issue.
  • Nice learning tool for new component owner
Other Resources

• Cadence Webinar – Testing the Testbench
  http://www.cadence.com/cadence/events/Pages/event.aspx?eventid=864

• SVUnit – a unit test framework in SystemVerilog

• CppUnit Cookbook