



UVM Cookbook

Recipe of the Month: C-Based Stimulus for UVM

Tom Fitzpatrick

Verification Evangelist

Design Verification Technology

October, 2012

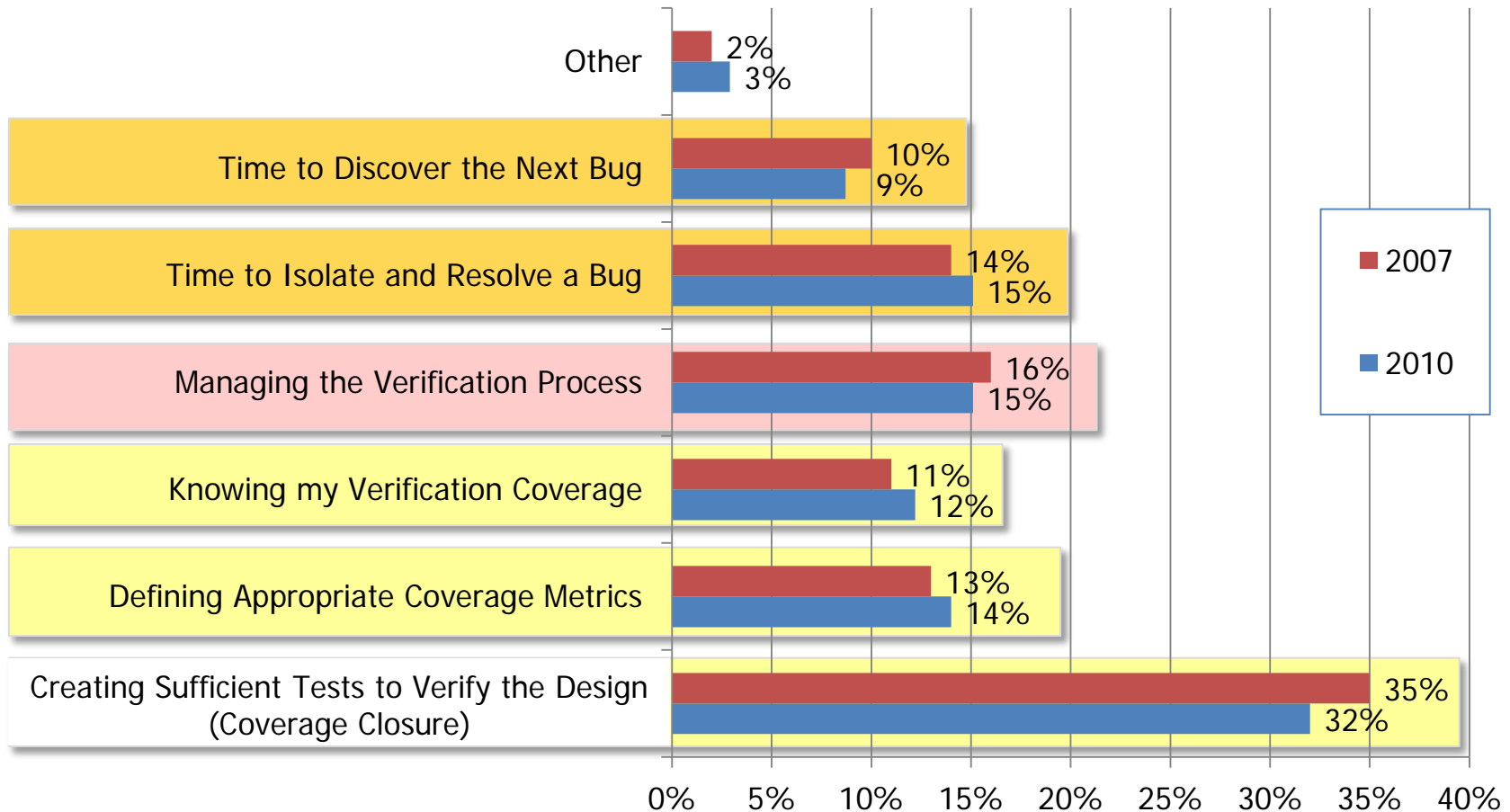
The Idea Behind The Methodology



- OVM & UVM underpin best practices
 - It's all about people...
 - Team Development
- Peopleware is most important
 - Develop Skill Set
 - Common language
 - Strategy and cohesion
 - Clarity and transparency
- A Guiding Methodology
 - Provides Freedom From Choice
 - Avoids Chaos and Repetition
 - Ease of Use APIs
 - Not just for Super-heroes!

Verification Challenges

Coverage , Process, & Debugging Still Identified as Top Priorities



Non-FPGA Designs

Source: Wilson Research Group and Mentor Graphics 2010 Functional Verification Study

How Do We Create More Tests?

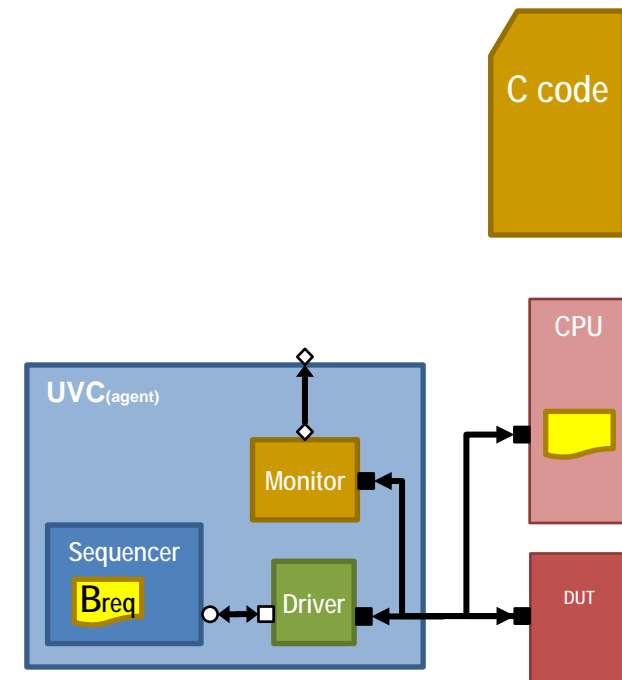


- UVM Sequences
 - Run with multiple seeds
 - Use factory to swap sequences
- inFact Intelligent Testbench Automation
 - Graph-based stimulus
 - Run as UVM Sequence
- Why Not C Code?
 - SW Engineers are available to write tests
 - Develop device driver code early
 - Reuse tests at higher levels of integration
 - Run on target device

How To Use C Code?

Option 1: Use CPU Model

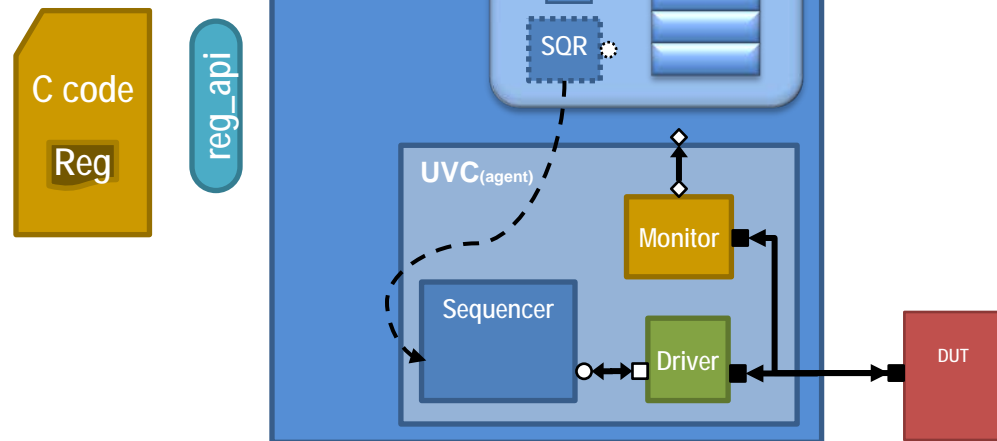
- Compile C code to run on processor model
- Instantiate processor along with DUT
- Use UVM to drive additional stimulus
 - Background traffic
- Pros
 - Visibility into CPU internals
- Cons
 - Setup overhead
 - Performance overhead



How To Use C Code?

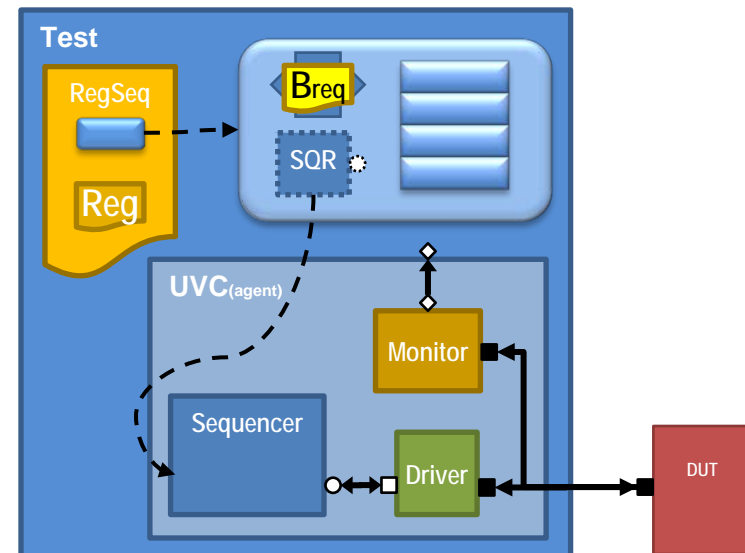
Option 2: Register R/W API + UVM Registers

- A C register read/write API
 - Use DPI to call SV tasks
 - SV tasks make UVM register accesses
 - C code runs on host workstation
- Pros
 - Lightweight
 - Simple extension to existing UVM environment
- Cons
 - Lack of CPU visibility



Quick Review: UVM Registers

- Registers contain fields
 - Fields contain bits
- Register block contains all registers that pertain to a DUT block
 - May also contain sub-blocks
- Register map specifies register offsets
 - Defines target sequencer
 - Defines register-to-bus adapter
- Register sequence executes register transactions

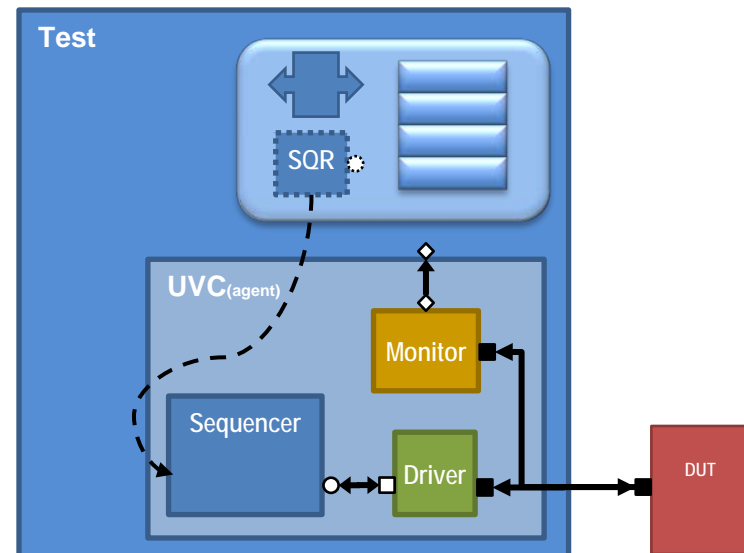


Quick Review: UVM Registers – The Env

```
class spi_env extends uvm_env;
  `uvm_component_utils(spi_env)
  apb_agent m_apb_agent;
  spi_env_config m_cfg;
  reg2apb_adapter reg2apb;
  uvm_reg_predictor #(apb_seq_item) apb2reg_predictor;

  function void build_phase(uvm_phase phase);
    uvm_config_db #(apb_agent_config)::set(this, "m_apb_agent*", "apb_agent_config",
      m_cfg.m_apb_agent_cfg);
    m_apb_agent = apb_agent::type_id::create("m_apb_agent", this);
    apb2reg_predictor =
      uvm_reg_predictor #(apb_seq_item)::
        type_id::create("apb2reg_predictor", this);
  endfunction:build_phase

  function void connect_phase(uvm_phase phase);
    reg2apb = reg2apb_adapter::
      type_id::create("reg2apb");
    if(m_cfg.spi_rm.get_parent() == null) begin
      m_cfg.spi_rm.APB_map.set_sequencer(
        m_apb_agent.m_sequencer, reg2apb);
    end
    apb2reg_predictor.map = m_cfg.spi_rm.APB_map;
    apb2reg_predictor.adapter = reg2apb;
    m_cfg.spi_rm.APB_map.set_auto_predict(0);
    m_apb_agent.ap.connect(apb2reg_predictor.bus_in);
  endfunction
endclass
```



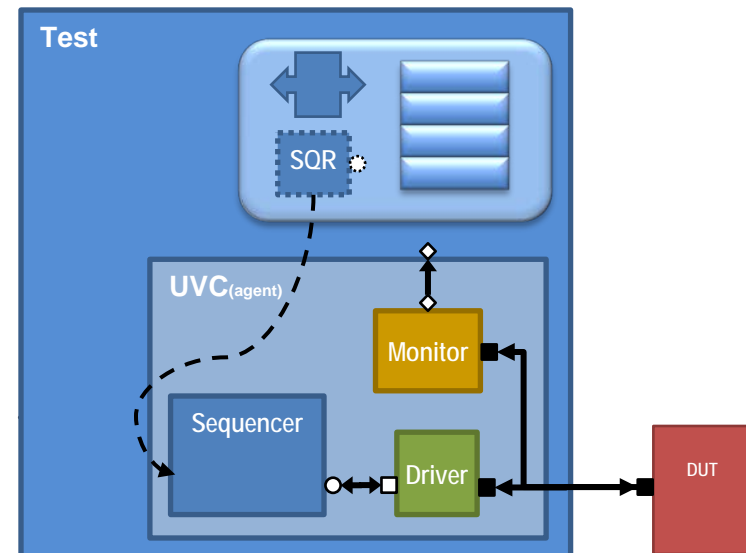
Quick Review: UVM Registers – Reg Block

```
class spi_reg_block extends uvm_reg_block;
  `uvm_object_utils(spi_reg_block)

  ...
  rand ctrl ctrl_reg;
  rand divider divider_reg;

  uvm_reg_map APB_map; // Block map

  virtual function void build();
    ...
    ctrl_reg = ctrl::type_id::create("ctrl");
    ctrl_reg.configure(this, null, "");
    ctrl_reg.build();
    ctrl_reg.add_hdl_path_slice("ctrl", 0, 14);
    divider_reg=divider::type_id::create("divider");
    divider_reg.configure(this, null, "");
    divider_reg.build();
    divider_reg.add_hdl_path_slice("divider", 0, 16);
    ...
    APB_map = create_map("APB_map", 'h0, 4,
                        UVM_LITTLE_ENDIAN);
    APB_map.add_reg(ctrl_reg, 32'h00000010, "RW");
    APB_map.add_reg(divider_reg, 32'h00000014, "RW");
    add_hdl_path("top_tb.DUT", "RTL");
    ...
  endfunction
```



Quick Review: UVM Registers – Sequence

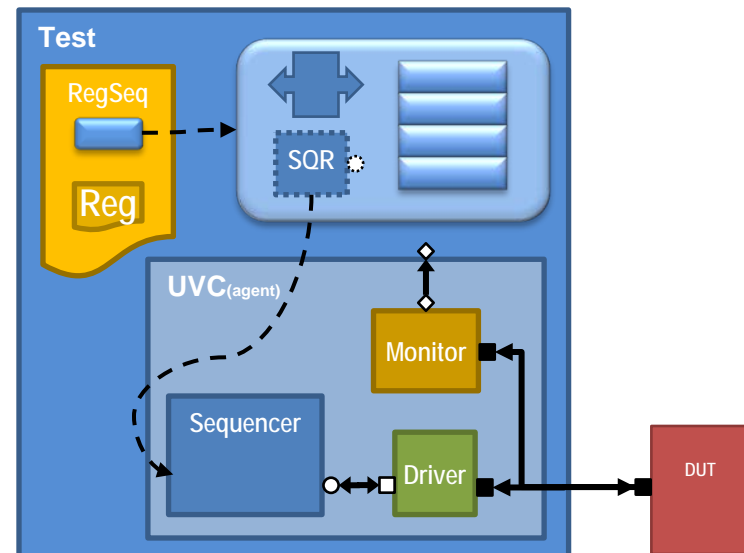
```
class send_10_chars_seq extends spi_bus_base_seq;
  `uvm_object_utils(send_10_chars_seq)
```

```
  uvm_reg_data_t control_config = 32'h2c30 ;
  uvm_reg_data_t data_0 = 32'hDEAD_BEEF,
                data_1 = 32'hBAAD_CAFE;
```

```
  task body;
    super.body();
    spi_rm.divider_reg.write(status, 2, .parent(this));
    spi_rm.ctrl_reg.write(status, control_config, .parent(this));
    spi_rm.ss_reg.write(status, 1, .parent(this));
    repeat(10) begin
      spi_rm.rxtx0_reg.write(status, data_0,
                            .parent(this));
      spi_rm.rxtx1_reg.write(status, data_1,
                            .parent(this));

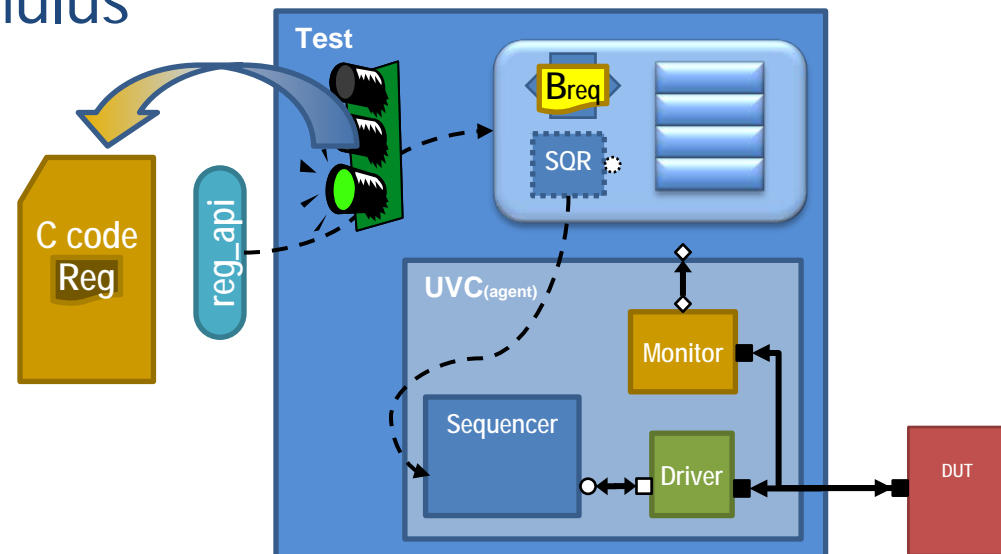
      spi_rm.ctrl_reg.write(status,
        (control_config + 32'h100), .parent(this));
      spi_rm.ctrl_reg.read(status, data,
        .parent(this));
      while(data[8] == 1) begin
        spi_rm.ctrl_reg.read(status, data,
          .parent(this));
      end
      data_0++;
      data_1++;
    end
  endtask: body
```

```
class spi_bus_base_seq
  extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(spi_bus_base_seq)
  spi_reg_block spi_rm;
  spi_env_config m_cfg;
  task body;
    if(!uvm_config_db #(spi_env_config)::get(null,
      get_full_name(), "spi_env_config", m_cfg)) begin
      `uvm_error("body", "Could not find spi_env_config")
    end
    spi_rm = m_cfg.spi_rm;
  endtask: body
```



C Stimulus Package Overview

- Requires a register model
- C code accesses registers via (addr,data)
 - Normal C code
 - ``include reg_api.h`
- Package uses DPI to convert to register reads/writes
- UVM test starts the C stimulus



An (Incredibly Simple) Example

simple_test.c

```
#include "spi_regs.h"
#include "reg_api.h"
```

```
int simple_test_routine() {
    int control = 0x2c30;
    int status;

    reg_write(CTRL, control);
    status = reg_read(CTRL);
}
```

```
int start_c_code() {
    simple_test_routine();
    return 0;
}
```

c_stimulus_pkg.sv

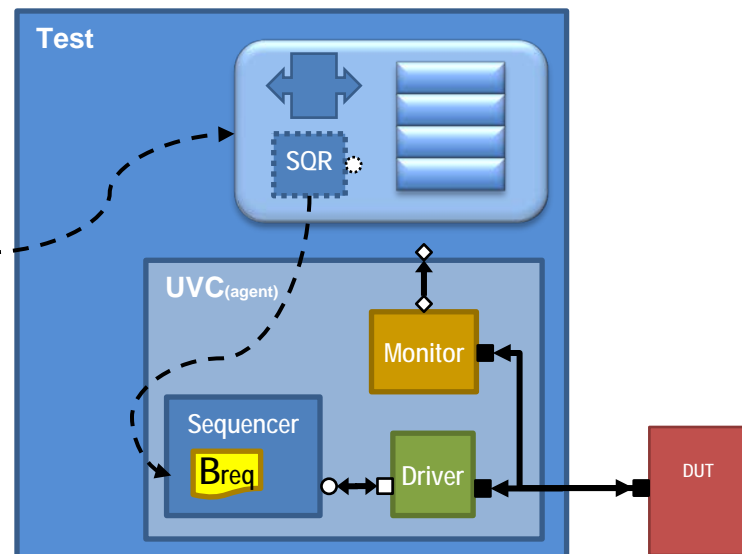
```
import "DPI-C" context task start_c_code();
```

simple_test.svh

```
task simple_test::run_phase(uvm_phase phase);
    phase.raise_objection(this, "Test Started");
    set_c_stimulus_register_block(spi_rm);
```

```
    fork
        start_c_code();
        bckgrnd_seq.start(env.spi.seqr);
    join_any
    phase.drop_objection(this, "Test Ended");
endtask
```

Imported C routine
called by SV



How It Works

c_stimulus_pkg.sv

```
task automatic c_reg_read(input int address, output int data);
```

```
...
endtask
```

```
task automatic c_reg_write(input int address, input int data);
```

```
uvm_reg_data_t reg_data;
```

```
uvm_status_e status;
```

```
uvm_reg write_reg;
```

```
write_reg = get_register_from_address(address);
```

```
if(write_reg == null) begin
```

```
`uvm_error("c_reg_write", $sformatf("Register not found at address: %0h", address))
```

```
return;
```

end

```
reg_data = data;
```

```
if(interrupt_in_progress == 1) begin
```

```
wait(interrupt_in_progress == 0);
```

end

```
write_reg.write(status, reg_data);
```

```
endtask: c_reg_write
```

```
export "DPI-C" task c_reg_write;
```

```
export "DPI-C" task c_reg_read;
```

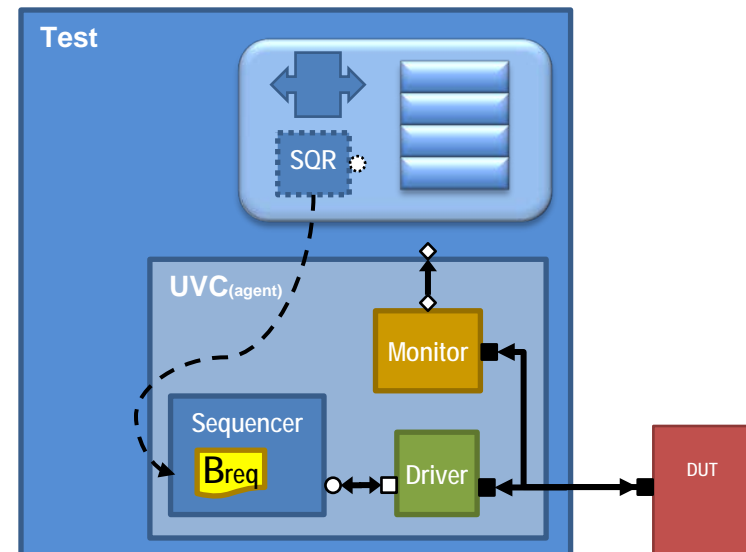
```
export "DPI-C" task wait_1ns;
```

```
import "DPI-C" context task start_c_code();
```

Exported SV tasks called by C



req_api



Back to the Example

reg_api.c

```
#include "reg_api.h"

int reg_read(int address) {
    int data;
    c_reg_read(address, &data);
    return data;
}

void reg_write(int address, int data) {
    c_reg_write(address, data);
}

void hw_wait_1ns(int n) {
    wait_1ns(n);
}

void register_thread() {
    svSetScope(svGetScopeFromName(
        "c_stimulus_pkg"));
}
```

c_stimulus_pkg.sv

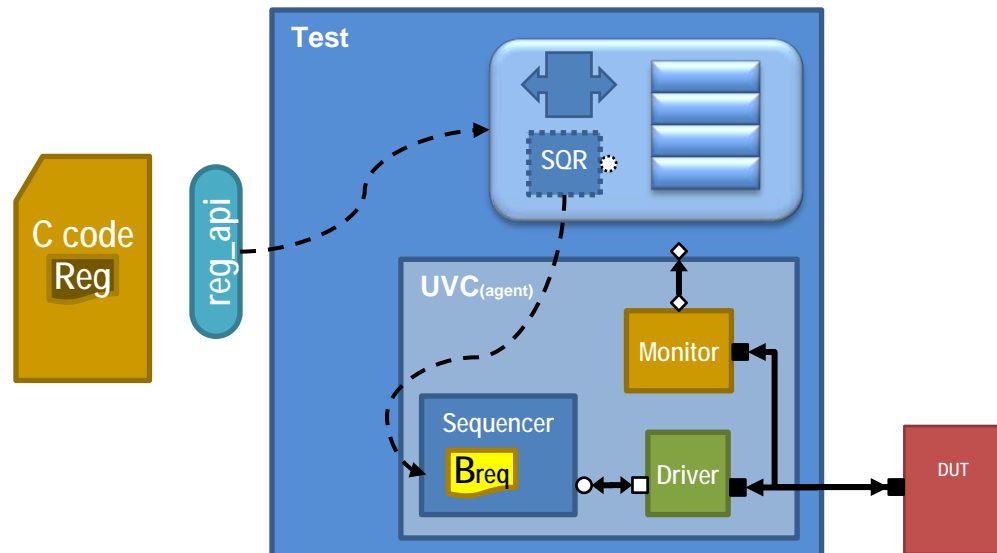
```
export "DPI-C" task c_reg_write;
export "DPI-C" task c_reg_read;
export "DPI-C" task wait_1ns;
import "DPI-C" context task start_c_code();
```

simple_test.c

```
#include "spi_regs.h"
#include "reg_api.h"

int simple_test_routine() {
    int control = 0x2c30;
    int status;

    reg_write(CTRL, control);
    status = reg_read(CTRL);
}
```



Side-by-Side

UVM Sequence

```
task body;
    super.body();
    repeat(10) begin
        spi_rm.rxtx0_reg.write(status, data_0,
                               .parent(this));
        spi_rm.rxtx1_reg.write(status, data_1,
                               .parent(this));
        spi_rm.ctrl_reg.write(status,
                               (control_config + 32'h100),
                               .parent(this));
        spi_rm.ctrl_reg.read(status, data, .parent(this));
        while(data[8] == 1) begin
            spi_rm.ctrl_reg.read(status, data,
                                  .parent(this));
        end

        data_0++;
        data_1++;
    end
endtask
```

C Code

```
int send_10_test_routine() {

    while(i < 10) {
        reg_write(TX0, data_0);

        reg_write(TX1, data_1);

        reg_write(CTRL, (control + 0x100));

        status = reg_read(CTRL);
        while((status & 0x100) == 0x100) {
            status = reg_read(CTRL);
        }
        i++;
        data_0++;
        data_1++;
    }
    return 0;
}
```

Adding Additional C Test(s)

new_test.c

```
#include "spi_regs.h"
#include "reg_api.h"
```

```
int new_test_routine() {
    int control = 0x2c30;
    int status;

    register_thread();

    status = reg_read(CTRL);
    ...
}
```

c_test_pkg.sv

```
`include "new_test.c"
import "DPI-C" context task new_test_routine();
```

new_test.svh

```
task new_test::run_phase(uvm_phase phase);
    phase.raise_objection(this, "Test Started");
    set_c_stimulus_register_block(spi_rm);
```

```
fork
```

```
    new_test_routine();
```

```
    bckgrnd_seq.start(env.spi.seqr);
```

```
join_any
```

```
    phase.drop_objection(this, "Test Ended");
```

```
endtask
```

Call new test explicitly

Non-default tests
must call
register_thread()

Import new C test
with context

Handling Interrupts – isr_pkg

int_test.c

```
#include "spi_regs.h"
#include "reg_api.h"

int int_test_routine() {
    int control = 0x2c30;
    int status;

    reg_write(CTRL, control);
    status = reg_read(CTRL);
}

int spi_isr() {
    ...
    status = reg_read(CTRL);
    reg_write(SS, 0x0);
    ...
}

int start_c_code () {
    int_test_routine();
    return 0;
}

int start_isr() {
    spi_isr();
    return 0;
}
```

Does **NOT** call
register_thread()

int_test.svh

```
task simple_test::run_phase(uvm_phase phase);
    phase.raise_objection(this, "Test Started");
    set_c_stimulus_register_block(spi_rm);

    fork
        start_c_code();
        bckgrnd_seq.start(env.spi.seqr);
    begin
        forever begin
            m_env_cfg.wait_for_interrupt();
            interrupt_service_routine();
        end
    end
    join_any
    phase.drop_objection(this, "Test Ended");
endtask
```

isr_pkg.sv

```
import c_stimulus_pkg::*;
task interrupt_service_routine;
    interrupt_in_progress = 1;
    start_isr();
    interrupt_in_progress = 0;
endtask: interrupt_service_routine

import "DPI-C" context task start_isr();
```

Compilation and Simulation

- Compile the `c_stimulus_pkg.sv` file, and if required, the `isr_pkg.sv`

```
vlog $(C_PKG_HOME)/c_stimulus_pkg.sv -dpiheader sv_dpi.h  
vlog $(C_PKG_HOME)/isr_pkg.sv -dpiheader sv_dpi.h
```

- Compile non-default C test routines in a test package:

```
vlog +incdir+$(TEST_PKG_HOME)  
$(TEST_PKG_HOME)/test_pkg.sv -dpiheader sv_dpi.h
```

- Compile the `reg_api.c` file

```
vlog +incdir+$(C_PKG_HOME) $(C_PKG_HOME)/reg_api.c
```

- Compile the C application c code

```
vlog +incdir+$(C_CODE_HOME) $(C_CODE_HOME)/my_c_code.c  
-ccflags -I$(C_PKG_HOME)
```

- Simulate

```
vsim top_tb +UVM_TESTNAME=spi_c_int_test
```

UVM C Stimulus Summary

- Provides a light-weight solution to allow C code to interact with the DUT
 - At the register level
- Allows for development of C code earlier in the project
- Lets software team contribute to test development
- Different from UVM Connect
 - Targeted only at allowing C code to communicate with Registers
 - Ideally, C code will be application-level
 - UVM Connect lets mixed-language components communicate
 - UVM Connect supports command-and-control across languages

Verification Academy—Provide the necessary skills that enable you to take advantage of today's latest advanced functional verification techniques.

Wide Variety of Topics

- *Verification Planning*
- *FPGA Verification*
- *Metrics for SoC Verification*
- *ABV*
- *Basic and Advanced UVM/OVM*
- *CDC*
- *UVM Express*
- *UVM Connect*
- *ADMS*
- *Intelligent Testbench Automation*
- *SystemVerilog Testbench Acceleration*

Verification Academy—Provide the skills that enable you to take advantage of the latest advanced functional verification

Wide Variety of Topics

- *Verification Planning* • *FPGA Verification*
- *Metrics for SoC Verification* • *ABV*
- *Basic and Advanced UVM/OVM* • *CDC*
- *UVM Express* • *UVM Connect* • *ADMS*
- *Intelligent Testbench Automation*
- *SystemVerilog Testbench Acceleration*

UVM/OVM Online Methodology Cookbook Available too!



The screenshot displays the Mentor Graphics Verification Academy interface. The top navigation bar includes 'Academy Home' and 'My Academy'. The main content area is titled 'Modeling Transactions' by Tom Fitzpatrick. It features a video player with a play button and a description of the session. Below the video, there are download links for various formats (MOV, MP3, MP4, PDF, HD) and a list of resources including a cookbook recipe, forum, and user contributions.

Mentor Graphics Verification Academy

Home > Academy Modules > UVM/OVM Verification > Advanced UVM > Modeling Transactions

Modeling Transactions
Tom Fitzpatrick

This session outlines the methods needed in the design of a sequence item (a.k.a. 'transaction') for use in UVM. It also discusses transaction extension a encapsulation to create more complex transactions.

Lecture **Rate**

Session Audience: walk Duration: 16 min

Separating Stimulus from the Testbench

- A key to reusability is to separate Behavior from Structure
- Transactions (a.k.a. Sequence Items) are the main communication vehicle across the boundary

views: 49
overall value: ★★★★★
your vote: ★★★★★

Resources:

- Cookbook Recipe: Sequences Items
- UVM Forum
- Accellera - UVM 1.1a
- UVM/OVM Online Methodology Cookbook
- UVM/OVM Kit Downloads & User Contributions

Download MOV 126.00 MB
Download MP3 14.30 MB
Download MP4 192.20 MB
Download PDF 1.06 MB
Download HD 294.00 MB

Verification Academy
skills that enable you to
latest advanced function

Wide Variety of

- Verification Planning
- Metrics for SoC Verification
- Basic and Advanced UVM
- UVM Express
- UVM Co
- Intelligent Testbench Automation
- SystemVerilog Testbench Automation

Article Comments

Welcome to the Mentor Graphics online *Methodology Cookbook* - An encyclopedia of verification knowledge.

Introduction:

- Read about the OVM, UVM and related verification best practices
- Follow one of our *guided tours* to learn about a topic from scratch
- Download supporting code examples tested on the latest Questa version
- Search recipes, patterns, techniques to solve your verification problem
- Participate in development by sending feedback, questions or contributions
- Download kit packages and utilities
- Create your own reference material to take away as PDF format books

Hot Topics...

- Testbench Overview
- Connections Overview
- Sequences Overview
- Analysis Overview
- End-Of-Test Overview

How to...

- Use a *guide*
- Download a *code example*
- Leave Comments or feedback
- Use the *search engine*
- Use the *category index*
- Ask an *expert*
- Find your way around

Category: Published

Privacy policy About Disclaimers Modified on 10 December 2010, at 09:44 - Accessed 5,580 times

Analysis/Overview - uvm

uvm.mentor.com/mc/Analysis/Overview

ga travel move read g-mark mgc mti org Other bookmarks

Mentor Graphics

Article Comments

Read Edit View history Search

Analysis/Overview

Works with Questa 2.1.1 Works with Questa 1.0

Contents (hide)

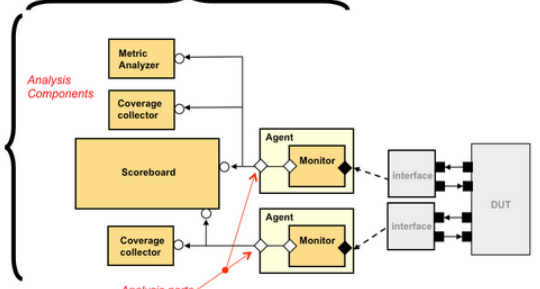
- 1 Overview
- 2 Monitoring DUT Activity
- 3 Scoreboards
- 4 Coverage Collectors
- 5 Metric Analyzers
- 6 Analysis Reporting

Overview

Verifying a design consists of two major parts: stimulus generation and an analysis of the design's response. Stimulus generation sets up the device and puts it in a particular state, then the analysis part actually performs the verification.

The analysis portion of a testbench is made up of components that observe behavior and make a judgement whether or not the device conforms to its specification. Examples of specified behavior include functional behavior, performance, and power utilization.

Analysis Layer



Monitoring DUT Activity

The process by which the analysis section makes its judgement starts with observing response activity in the device under test (DUT). This is done by one or more monitors that observe the signal-level activity on the DUT through a virtual interface(s). The monitor converts signal-level activity into TUI transactions, and broadcasts the transactions to interested analysis components using *analysis ports* which are connected to subscribers. These subscribers capture the transactions and perform their analysis.

Scoreboards

These analysis components collect the transactions sent by the monitor and perform specific analysis activities on the collection of transactions. Scoreboard components determine whether or not the device is functioning properly. The best scoreboard architecture separates its tasks into two areas of concern: prediction and evaluation.

A *predictor model*, sometimes referred to as a "golden Reference Model", receives the same stimulus stream as the DUT and produces known good response transaction streams. The scoreboard evaluates the predicted activity with actual observed activity on the DUT.

A common evaluation technique when there is one expected stream and one actual stream is to use a comparator, which can either compare the transactions assuming in-order arrival of transactions or out-of-order arrival.

Coverage Collectors

These analysis components capture the transactions sent by the monitor and perform specific analysis activities on the collection of transactions. Coverage collectors determine whether or not the device is functioning properly. The best coverage collector architecture separates its tasks into two areas of concern: prediction and evaluation.

A *predictor model*, sometimes referred to as a "golden Reference Model", receives the same stimulus stream as the DUT and produces known good response transaction streams. The coverage collector evaluates the predicted activity with actual observed activity on the DUT.

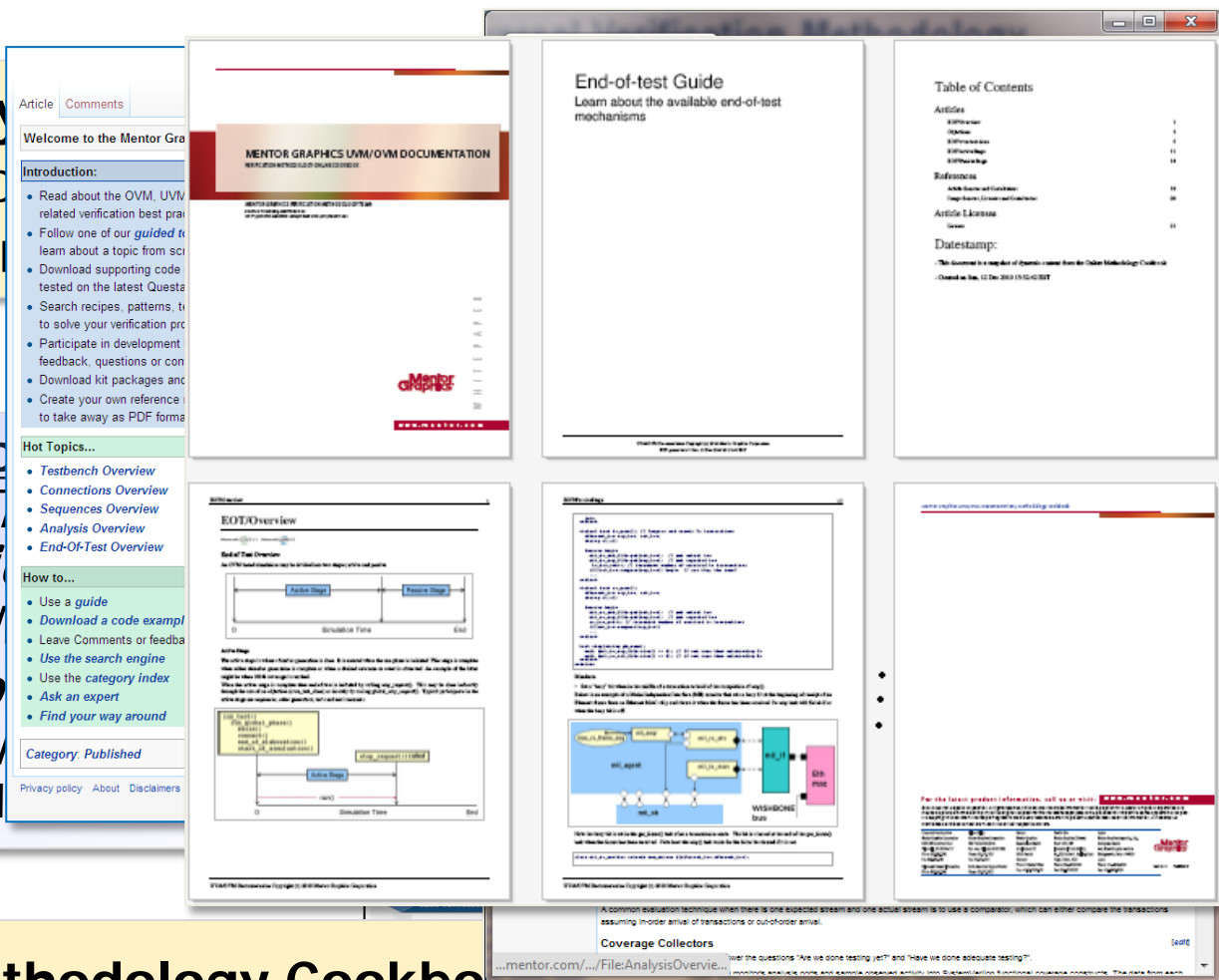
A common evaluation technique when there is one expected stream and one actual stream is to use a comparator, which can either compare the transactions assuming in-order arrival of transactions or out-of-order arrival.

UVM/OVM Online Methodology Cookbook Available too:

Verification Academy
skills that enable you to
latest advanced function

Wide Variety of

- *Verification Planning*
- *Metrics for SoC Verification*
- *Basic and Advanced UVM*
- *UVM Express*
- *UVM Co*
- *Intelligent Testbench Automation*
- *SystemVerilog Testbench Automation*



The screenshot displays the Mentor Graphics UVM/OVM Documentation website. It features a sidebar with navigation links such as 'Welcome to the Mentor Graphics UVM/OVM Documentation', 'Introduction', 'Hot Topics...', 'How to...', and 'Category: Published'. The main content area shows several articles, including 'End-of-test Guide', 'Table of Contents', 'EUT Overview', 'Wishbone Bus', and 'Coverage Collectors'. Each article includes a brief description and a link to the full document.

UVM/OVM Online Methodology Cookbook Available too:



www.mentor.com