Coding Guidelines and Code Generation

John Aynsley · Doulos · john.aynsley@doulos.com / **Dr. Christoph Sühnel · Doulos** · christoph.suehnel@doulos.com

Introduction

Abstract

Easier UVM consists of a comprehensive set of coding guidelines for the use of UVM and an open-source UVM code generation tool that automatically generates the boilerplate UVM code for a project according to these guidelines.

Easier UVM helps individuals and teams get started with UVM, helps avoid pitfalls, helps promote best practice, and helps ensure consistency and uniformity across projects.

Easier UVM helps teams to become productive with UVM more quickly, and reduces the burden of maintaining a UVM codebase over time. Both the guidelines and the tool can be taken as they are or can be used as a starting point and modified according to the demands of a specific project.

Motivation

- SystemVerilog is large and complex
- Differences between simulators
- UVM is large and complex
- There's More Than One Way To Do It!
- New users don't know where to start

Benefits

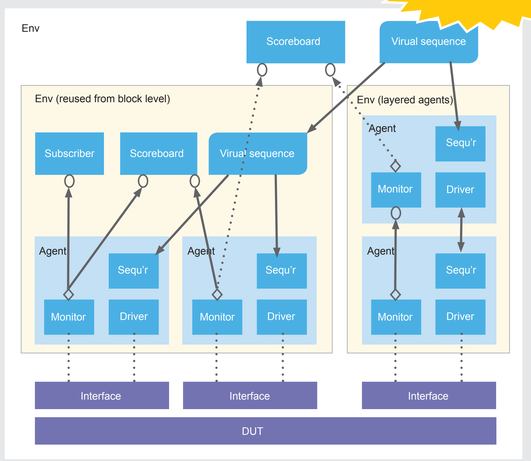
- Helps getting started
- Learn best practice and avoid common pitfalls
- Become productive more quickly
- Be uniform and consistent across projects
- Reduces support costs over time

Visit Doulos at **Booth 801**

www.doulos.com

Coding Guidelines

General Guidelines



- Tests
- Reuse
- The factory and the configuration database
- Transaction-level ports and exports
- Virtual interfaces
- Run-time phases
- Virtual sequences and scoreboards
- Message ID and verbosity
- Register layer
- Functional coverage
- Structuring files

Coding Guidelines

- Lexical Guidelines and Naming Conventions
- General Guidelines
- General Code Structure
- Clocks, timing and synchronization
- Transactions
- Sequences
- Objections
- Components
- Connection to the DUT
- TLM Connections
- Configurations
- The Factory
- Tests
- Messaging
- Functional Coverage
- The Register Layer
- Agent Data Structure and Packaging

Mostly common sense

More prescriptive than UVM docs

Coding Patterns

Pattern 1

```
class my_comp extends uvm_component;
  `uvm_component_utils(my_comp)

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(...);
    ...
endclass
```

Pattern 2a

```
class my_tx extends uvm_sequence_item;
  `uvm_object_utils(my_tx)

  function new (string name = "");
    super.new(name);
  endfunction

  function string convert2string;
    ...
endclass
```

Pattern 2b

```
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

  function new(string name = "");
    super.new(name);
  endfunction
  ...
  task body;
    ...
endclass
```

Code Generation

Code Generation – INPUT

For each DUT interface, you specify

- Agent name

```
uvc_Name| spi
```

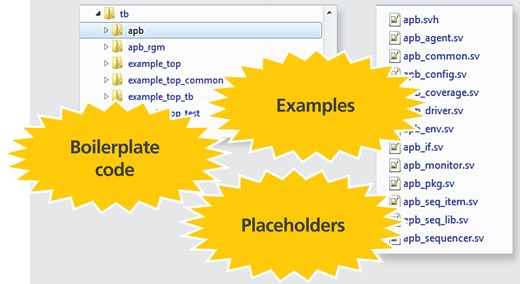
- Sequence item name and list of variables

```
uvc_item | spi_seq_item
uvc_var | rand logic [127:0] data;
uvc_var | rand bit [6:0] no_bits;
uvc_var | rand bit RX_NEG;
```

- Interface name and list of clocks, resets & variables

```
uvc_if | spi_if
uvc_port | logic clk;
uvc_port | logic reset;
uvc_port | logic sclk_pad_o;
...
```

Code Generation – OUTPUT



Example

```
task spi_driver::run_phase(uvm_phase phase);
  // add additional declarations here
  super.run_phase(phase);
  `uvm_info(get_type_name(),"run_phase",UVM_MEDIUM)
  // set signals on reset values here
  @(posedge vif.reset) // reset goes inactive
  forever begin
    seq_item_port.get_next_item(req);
    @(posedge vif.clk)
    `uvm_info(get_type_name(),
      {"req item\n",req.sprint}, UVM_MEDIUM)
    // insert the driver protocol here
    $cast(rsp, req.clone());
    // adopt the rsp
    seq_item_port.item_done();
  end
endtask : run_phase
```

Practical Experience

1. Kick-off meeting
2. Create setup files
3. Generate code for complete environment
4. Simulate complete environment
5. Implement drivers one-by-one
6. Simulate each driver by adding new sequences and tests
7. Implement monitors, subscribers, and scoreboards
8. Add further data members and refine methods