**Introduction**

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!
- UVM is large and complex
- Differences between simulators

**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

**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.

**Coding Guidelines**

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

**Coding Patterns**

**Pattern 1**

```plaintext
class my_comp extends uvm_component;
  `uvm_component_utils
  uvm_component_utils(my_comp);
  function new(name = "");
    super.new(name, parent);
    endfunction
  function void build_phase(...);
    endclass
```

**Pattern 2a**

```plaintext
class my_seq extends uvm_sequence;
  `uvm_sequence_item_utils
  uvm_sequence_item_utils(my_seq);
  function new(name = "");
    super.new(name);
    endfunction
  function void body(...);
    endclass
```

**Pattern 2b**

```plaintext
class my_seq extends uvm_sequence_item;
  `uvm_object_utils
  uvm_object_utils(my_seq);
  function new(name = "");
    super.new(name);
    endfunction
  task body;
    endclass
```

**Code Generation**

**Code Generation – INPUT**

For each DUT interface, you specify:

- Agent name
- Sequence item name and list of variables
- Virtual sequences and scoreboards
- Message ID and verbosity
- Register layer
- Functional coverage
- Structuring files

**Code Generation – OUTPUT**

**Example**

```plaintext
task spi_driver::run_phase(uvm_phase phase);
    // insert the driver protocol here
    body自然界

    // test signals on reset values here
    $cast(spi_if, req.Clone());
    // add additional declarations here
    super.run_phase(phase);
    `uvm_info(get_type_name(), "run_phase", UVM_MEDIUM)
```

**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. Implement each driver by adding new sequences and tests
7. Implement monitors, subscribers, and scoreboards
8. Add further data members and refine methods

Visit Doulos at Booth 801

**www.doulos.com**