Agnostic UVM-XX Testbench Generation

Jacob Andersen, SyoSil ApS, Copenhagen, Denmark

Stephan Gerth, Fraunhofer IIS, Dresden, Germany

Filippo Dughetti, SyoSil ApS, Copenhagen, Denmark

accellera SYSTEMS INITIATIVE

SyoSil Systems on Silicon

Fraunhofer IIS

2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

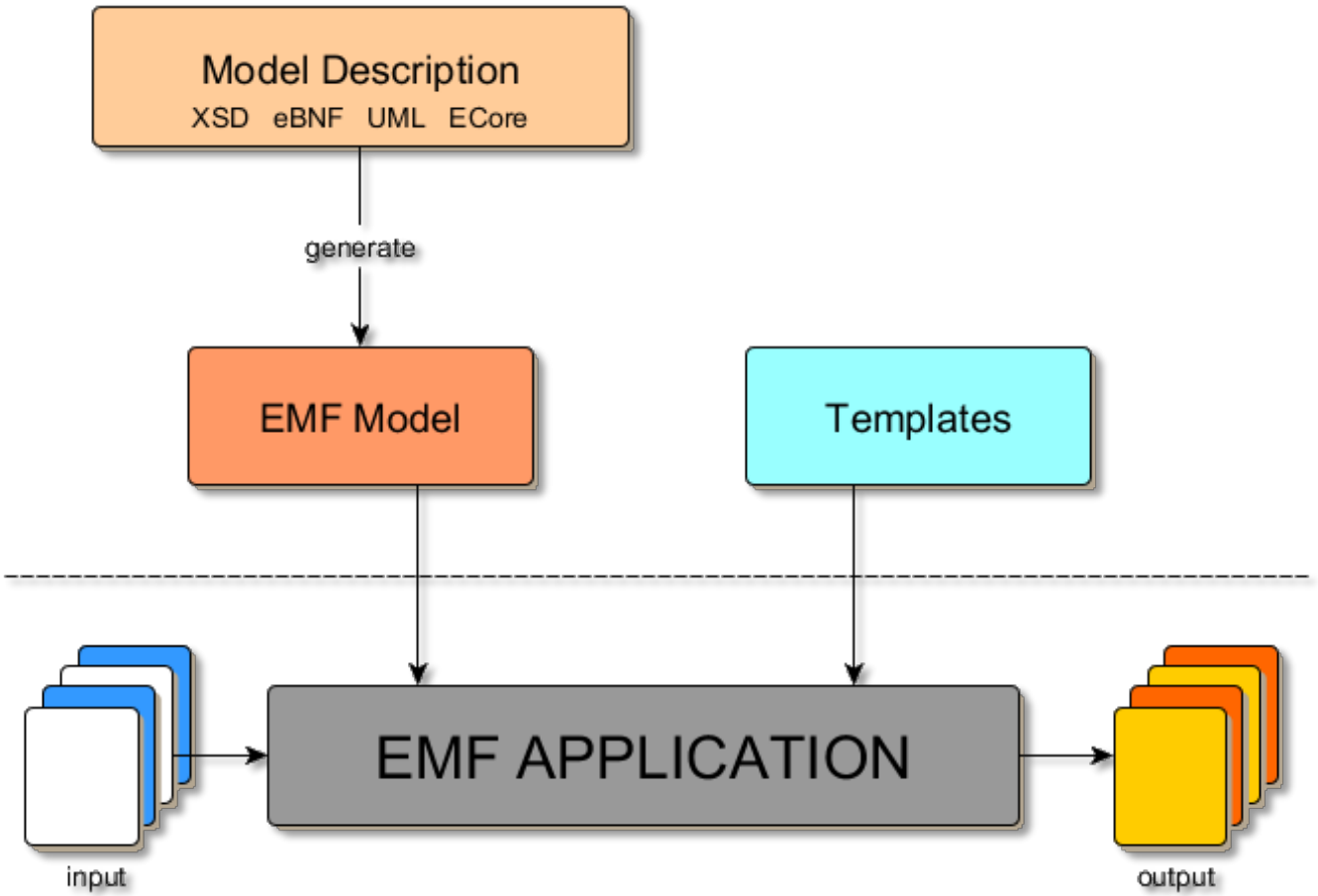
Introduction

- Code generation and **model driven software development** present for long time in ASIC verification
- **Simplify** usage of UVM **by abstraction** of the testbench
- Generate SystemVerilog and SystemC testbenches
- Reusable UVM Verification Components (UVCs) and testbenches supporting hardware in the loop (HiL)

Eclipse Modeling Framework

- UVM-XX generator implemented in the open source **Eclipse modeling Framework (EMF)**
- EMF benefits:
  - Preservable **user regions**
  - Easiness of updating the abstract specification
  - Multiple input formats
  - Multiple output formats via simple templates
  - Eclipse context aware editing and syntax highlighting

EMF Flow


Model Description
XSD eBNF UML ECore
generate
EMF Model
Templates
EMF APPLICATION
input
output

UVM Testbench Abstraction

- Pragmatism when defining the abstraction
- Layered abstraction:
  - **LUVC**: implementation of UVCs
  - **LB2B**: implementation of back2back testbench
  - **LTB**: implementation of a RTL testbench
- Common **Domain Specific Language (DSL)** for all three layers and their relations
  - compact, easy to read format
  - Verification Environment format (VE)

UVM Testbench Abstraction

- EMF supports XTEXT for defining an eBNF-like specification of the VE DSL
- Single DSL used for all three layers: a separation on syntax level is needed
- Two types: VerificationComponent (LUVC) or EnvironmentComponent (LB2B, LTB).

```
Component:
  VerificationComponent | EnvironmentComponent
;
VerificationComponent:
  'type' '=' '"vc"'
  'name' '=' name = STRING
  …
EnvironmentComponent:
  (imports+=Import)*
  'type' '=' '"env"'
  'name' '=' name = STRING
  'mode' '=' mode = STRING
  …
```

UVM Testbench Abstraction

- Finding the correct level of abstraction versus expressiveness for specifying a UVC is the key

```
VerificationComponent:
        'type' '=' '"vc"'
        'name' '=' name = STRING
        ( constantBlocks += ConstantBlock
          | typeBlocks += TypeBlock
          | agentBlocks += AgentBlock
          | driverBlocks += DriverBlock
          | interfaceBlocks += InterfaceBlock
          | checkBlocks += CheckBlock
          | transactionBlocks += TransactionBlock
          | configurationBlocks += ConfigurationBlock)*
;
```

- Protocol specific code and types handling

UVM Testbench Abstraction

- Type handling can get difficult when targeting SytemVerilog AND SystemC
- Three ways to support types
  - Raising the abstraction level of types
  - Full SV to SC type translator which would parse the existing SV string and translate that into the appropriate SC type
  - Simpler version of (2) with fixed mapping of standard types
- Addition of an optional sc_type string

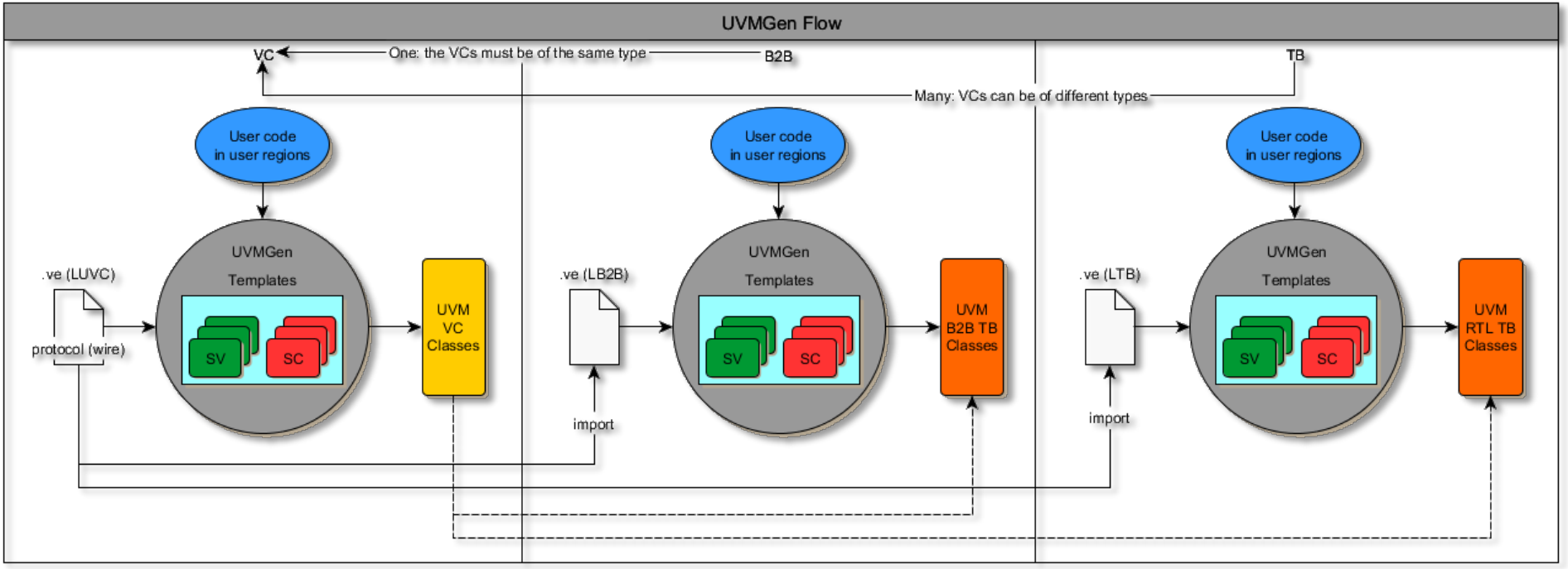
UVM Testbench Abstraction

- EnvironmentComponent defines information needed for generation of LB2B and LTB layers

```
EnvironmentComponent:
        (imports+=Import)*
        'type' '=' '"env"'
        'name' '=' name = STRING
        'mode' '=' mode = STRING
        ( verifcompBlocks += VerifcompBlock
          | scoreboardBlocks += ScoreboardBlock
          | envconfigBlocks += EnvconfigBlock
          | sequenceBlocks += SequenceBlock
          | testBlocks += TestBlock)*
;
Import:
        'import' importURI = STRING
;
```

Generating UVM

- UVM code generation is straight forward due to the EMF framework



Wishbone Example - UVC

- Wishbone UVC
- VE file with type field set to “vc” has been defined

```
//-----------------------------
// Definition for the WB VC
// Lead UVMGen example.
//-----------------------------
type="vc"
name="wb"
```

Wishbone Example - UVC

- VE file continued with remaining blocks as defined by the DSL

```
//-------------------------------------------------------------------------
// Wishbone protocol interface definition
//-------------------------------------------------------------------------
agent {  numInst = 1; protocol = wishbone;  }
driver wishbone {
  variant = "master"          comment="Master side of protocol";
  variant = "slave"           comment="Slave side of protocol"; }
interface {
  parameter {
    name="WB_DATA_BYTE_NUM" type="int" value="pk_wb::WB_DATA_BYTE_NUM"
      comment="WB data number of bytes";
    name="WB_DATA_WIDTH"    type="int" value="pk_wb::WB_DATA_WIDTH"
      comment="WB data width";
    …  }
  clock {  name="clk"     type="logic"  comment="Main clock";  }
  reset {  name="rst"     type="logic"  comment="Main reset";  }
  signal {
    name="dat_o" type="logic[WB_DATA_WIDTH-1:0]"    driver="master"
      resetval="'x"  comment="WISHBONE data with same direction as adr";
    name="adr"   type="logic[WB_ADDR_WIDTH-1:0]"    driver="master"
      resetval="'x"  comment="WISHBONE address";
    name="sel"   type="logic[WB_DATA_BYTE_NUM-1:0]" driver="master"
      resetval="'x"  comment="WISHBONE byte select";   …  }
}
```

Wishbone Example - UVC

- UVMGen generates the UVM classes for the UVC

```
class cl_wb_wishbone_agent extends uvm_agent;
//------------------------------------------------------------------------------
  // Analysis port connected to monitor
  uvm_analysis_port #(cl_wb_seq_item) ap;
  cl_wb_config cfg;           // Handle to configuration object
  cl_wb_sequencer sequencer;  // Transaction sequencer
  cl_wb_wishbone_monitor monitor;      // Signal monitor
  cl_wb_wishbone_driver driver;        // Signal driver
  // ************** KEEP ->Start of user code FIELDS
  // Add user code for FIELDS here!
  // ************** KEEP ->End of user code FIELDS
  …
// Builds the agent
//------------------------------------------------------------------------------
function void cl_wb_wishbone_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);
  // Contruct analysis port
  this.ap = new("ap", this);
  // if no cfg available from global table, a random one is generated
  if (!uvm_config_db #(cl_wb_config)::get(this, "", "cfg", this.cfg))
begin
    `uvm_warning("CFG", {"Configuration object not initialized from ",
      "outside. Generating one internally"});
    this.cfg = cl_wb_config::type_id::create("cfg");
    if(!this.cfg.randomize()) begin
      `uvm_fatal("CONFIG", "Unable to randomize configuration");
    end
  end
  // Creates the monitor, a handle to 'cfg' is passed down to the monitor
  uvm_config_db #(cl_wb_config)::set(this, "monitor", "cfg", this.cfg);
  this.monitor = cl_wb_wishbone_monitor::type_id::create("monitor",this);
  if(this.cfg.is_active) begin
    // Creates the driver (conditionally: 'active_passive')
    // a handle to 'cfg' is passed down to the driver
    uvm_config_db #(cl_wb_config)::set(this, "driver", "cfg", this.cfg);
    this.driver = cl_wb_wishbone_driver::type_id::create("driver", this);
```

```
class cl_wb_wishbone_agent: public uvm::uvm_agent {
//------------------------------------------------------------------------------
  // Analysis port connected to monitor
  uvm_analysis_port<cl_wb_seq_item>* ap;
  cl_wb_config* cfg;           // Handle to configuration object
  cl_wb_sequencer* sequencer;  // Transaction sequencer
  cl_wb_wishbone_monitor* monitor;      // Signal monitor
  cl_wb_wishbone_driver* driver;        // Signal driver
  // ************** KEEP ->Start of user code FIELDS
  // Add user code for FIELDS here!
  // ************** KEEP ->End of user code FIELDS
  …
// Builds the agent
//------------------------------------------------------------------------------
  void build_phase(uvm::uvm_phase phase) {
    uvm::uvm_agent::build_phase(phase);
    // Contruct analysis port
    ap = new("ap", this);
    // if no cfg available from global table, a random one is generated
    if (!uvm::uvm_config_db<cl_wb_config>::get(this, "", "cfg", cfg)) {
      UVM_WARNING("CFG", "Configuration object not initialized from
outside. Generating one internally");
      cfg = cl_wb_config::type_id::create("cfg");
      if(!cfg->randomize()) {
        UVM_FATAL("CONFIG", "Unable to randomize configuration");
      }
    }
  // Creates the monitor, a handle to 'cfg' is passed down to the monitor
    uvm::uvm_config_db<cl_wb_config>::set(this, "monitor", "cfg", cfg);
    monitor = cl_wb_wishbone_monitor::type_id::create("monitor", this);
    if(is_active) {
      // Creates the driver (conditionally: 'active_passive')
      // a handle to 'cfg' is passed down to the driver
      uvm::uvm_config_db<cl_wb_config>::set(this, "driver", "cfg", cfg);
    driver = cl_wb_wishbone_driver::type_id::create("driver", this);
  }
};
```

Wishbone Example – B2B TB

- Similar approach for generating a back to back testbench

```
//-------------------------------------------------------
// Definition for the WB back2back verification setup.
// Lead UVMGen example.
//-------------------------------------------------------
import "wishbone.ve"
type="env"
name="wb_b2b"
mode="back2back"
```

Wishbone Example – B2B TB

- VE file continued with remaining blocks as defined by the DSL

```
//------------------------------------------------------------------------
// List of verification components to use
// Type must be known VC
//------------------------------------------------------------------------
verifcomp {
  name="wb_master"  type="wb"  active="yes" driver="master";
  name="wb_slave"   type="wb"  active="yes" driver="slave";
  name="wb_monitor" type="wb"  active="no" ; }
//------------------------------------------------------------------------
// List of configurations to be used by tests
//------------------------------------------------------------------------
envconfig {
   name="basicconf"   base="";           // Declares cl_config_basicconf,
extends cl_wb_b2b_config
   name="userconf"    base="basicconf"; }// Declares cl_config_userconf,
extends cl_config_basic

...
```

Wishbone Example – B2B TB

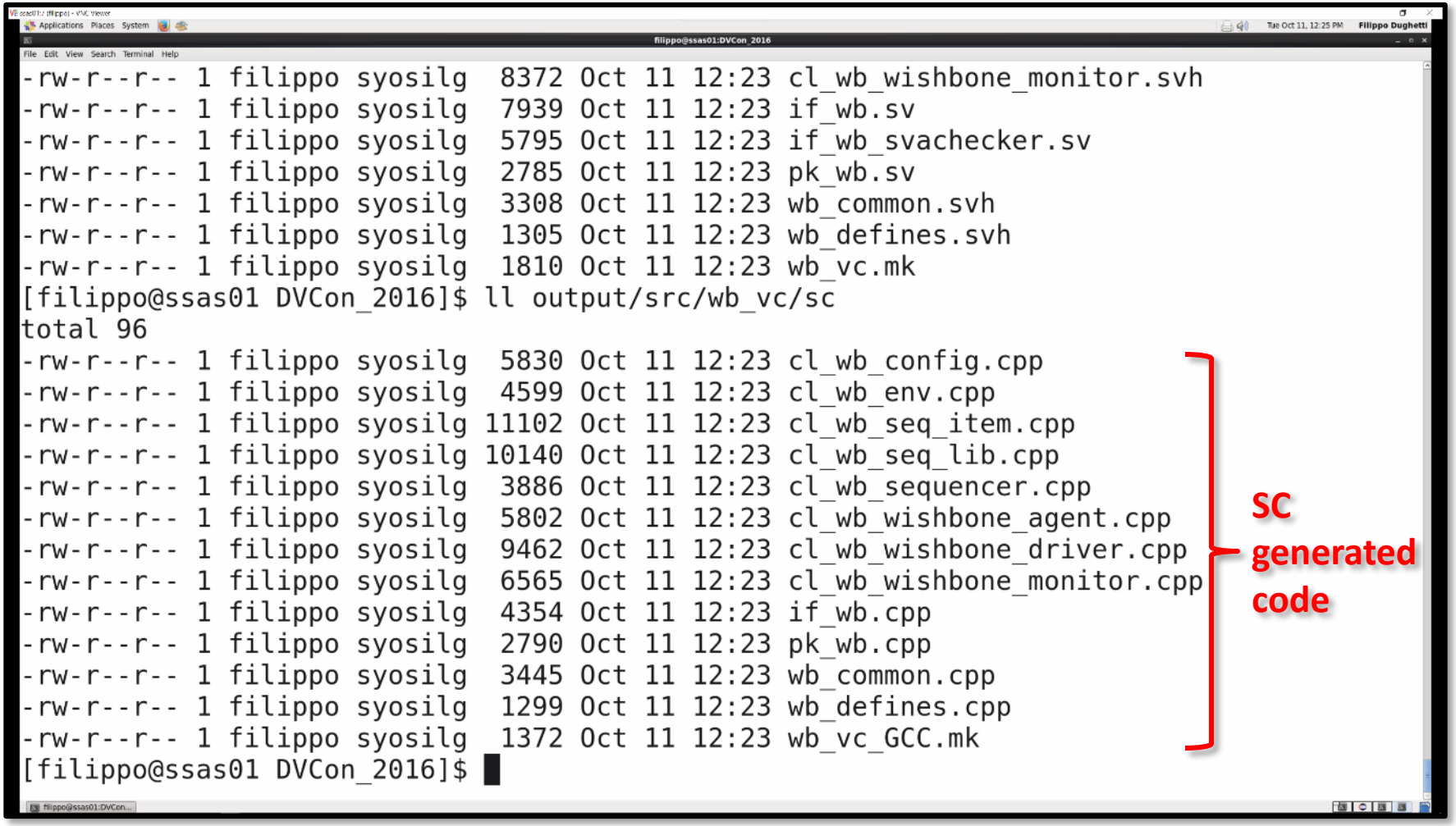
- UVMGen generates the UVM classes for the B2B TB

```
class cl_wb_b2b_env extends uvm_env;
//------------------------------------------------------------------------
  cl_wb_b2b_config cfg; // Configuration class handle
  // Virtual sequencer
  cl_wb_b2b_virtual_sequencer virtual_sequencer;
  // Module VC(s) & Interface VC(s)
  cl_wb_env wb_master;
  cl_wb_env wb_slave;
  cl_wb_env wb_monitor;
  // ************** KEEP ->Start of user code FIELDS
  // Add user code for FIELDS here!
  // ************** KEEP ->End of user code FIELDS
  …
// Build environment
//------------------------------------------------------------------------
function void cl_wb_b2b_env::build_phase(uvm_phase phase);
  pre_build();
  super.build_phase(phase);
  // If no cfg available from global table, a random one is generated
  if (!uvm_config_db #(cl_wb_b2b_config)::get(this, "", "cfg",
this.cfg)) begin
    `uvm_warning("CFG", {"Configuration object not initialized from ",
      "outside. Generating one internally"});
    this.cfg = cl_wb_b2b_config::type_id::create("cfg");
    if(!this.cfg.randomize()) begin
      `uvm_fatal("CONFIG", "Unable to randomize configuration");
    end
  end

  uvm_config_db #(cl_wb_b2b_config)::set(this, "virtual_sequencer",
"cfg", this.cfg); // push down config
  this.virtual_sequencer =
cl_wb_b2b_virtual_sequencer::type_id::create("virtual_sequencer",
this);
  uvm_config_db #(cl_wb_config)::set(this, "wb_master", "cfg",
this.cfg.wb_master_cfg);
  this.wb_master = cl_wb_env::type_id::create("wb_master", this);
  uvm_config_db #(cl_wb_config)::set(this, "wb_slave", "cfg",
this.cfg.wb_slave_cfg);
  this.wb_slave = cl_wb_env::type_id::create("wb_slave", this);
```

```
class cl_wb_b2b_env: public uvm::uvm_env {
//------------------------------------------------------------------------
public:
  cl_wb_b2b_config* cfg; // Configuration class handle
  // Virtual sequencer
  cl_wb_b2b_virtual_sequencer* virtual_sequencer;
  // Module VC(s) & Interface VC(s)
  cl_wb_env* wb_master;
  cl_wb_env* wb_slave;
  cl_wb_env* wb_monitor;
  // ************** KEEP ->Start of user code FIELDS
  // Add user code for FIELDS here!
  // ************** KEEP ->End of user code FIELDS
  …
// Build environment
//------------------------------------------------------------------------
  void build_phase(uvm::uvm_phase& phase) {
    pre_build();
    uvm::uvm_env::build_phase(phase);
    // If no cfg available from global table, a random one is generated
    if (!uvm::uvm_config_db<cl_wb_b2b_config*>::get(this, "", "cfg",
cfg)) {
      UVM_WARNING("CFG", "Configuration object not initialized from
outside. Generating one internally");
      cfg = cl_wb_b2b_config::type_id::create("cfg");
      if(!this.cfg.randomize()) {
        UVM_FATAL("CONFIG", "Unable to randomize configuration");
      }
    }
    uvm::uvm_config_db<cl_wb_b2b_config>::set(this, "virtual_sequencer",
"cfg", cfg); // push down config
    virtual_sequencer =
cl_wb_b2b_virtual_sequencer::type_id::create("virtual_sequencer", this);
    uvm::uvm_config_db<cl_wb_config*>::set(this, "wb_master", "cfg",
cfg.wb_master_cfg);
    wb_master = cl_wb_env::type_id::create("wb_master", this);
    uvm::uvm_config_db<cl_wb_config*>::set(this, "wb_slave", "cfg",
cfg.wb_slave_cfg);
    wb_slave = cl_wb_env::type_id::create("wb_slave", this);
    uvm::uvm_config_db<cl_wb_config*>::set(this, "wb_monitor", "cfg",
```

Demo



Future Extensions

- Extend this to handle LTB as well
- Abstraction of functional coverage
- Extension of UVMGen limitless as template mechanism provides everything necessary
  - Example: testbench documentation could be generated from the same EMF model including a figure showing the testbench architecture, since the EMF model contains this information

Questions