Agnostic UVM-XX Testbench Generation

Replace XX (almost) as you see fit!

Jacob Andersen, SyoSil ApS, Copenhagen, Denmark (*jacob@syosil.com*)

Stephan Gerth, Fraunhofer IIS, Dresden, Germany (*stephan.gerth@eas.iis.fraunhofer.de*)

Filippo Dughetti, SyoSil ApS, Copenhagen, Denmark (*filippo@syosil.com*)

***Abstract*—Code generation or model driven software development has always had his place within the field of ASIC verification due to the obvious advantages with respect to time savings, complexity reduction, less bugs/errors etc. Typically, model driven software development has been used for generating RTL implementation for registers, register documentation, self-contained register tests from abstract specifications such as IP-XACT. Over the last couple of years generation of testbenches implemented in UVM have been widely introduced within the field by several contributors. This paper tries to leverage all of this previous work and introduce a layered abstraction for UVM testbenches which makes it possible to generate UVM-SystemVerilog (UVM-SV) and UVM-SystemC (UVM-SC) based testbenches from the same abstract specification. Especially UVM-SystemC enables the reuse of testbenches, e.g. from concept level down to Hardware-in-the-loop (HiL) approaches.**

Keywords—SytemVerilog; UVM; UVM-SC; SystemC; Testbench Generation; Constrained Random Verification

I. Introduction

A. Setting the scene

In many scenarios code generation or model driven software development can be beneficial. Especially, if one can reduce the complexity greatly by introducing an appropriate abstraction and then generate the actual source code. Non-ASIC verification software developers have been using this technique for decades.

With the introduction of Universal Verification Methodology (UVM) [1], Application Specific Integrated Circuit (ASIC) verification entered that scenario since UVM testbenches are structured by nature and many parts can be generated. This paper presents an abstraction of a UVM testbench to generate both, SystemVerilog [2] (UVM-SV) and SystemC [3] (UVM-SC) testbenches. It also introduces new features compared to existing UVM generators by exploiting model driven software development.

This paper will describe the extensions needed to support generating UVM-SC code on an existing UVM generator capable of generating UVM-SV code. Furthermore, all generated code will be based on a single example using the Wishbone protocol [4].

B. Benefits for the users and the UVM community

Existing UVM generators are matched by this generator since it is also able to generate UVM-SV-based testbenches. Additionally, it will aid people in writing reusable UVM Verification Components (UVCs) and testbenches supporting hardware in the loop (HiL) as described in [5]. A reference platform for the UVM-SC implementation will also be provided, as well as a Wishbone-based reference example: this can potentially be explored by all IPs available from OpenCores [6]. Furthermore, the ability to validate UVCs implemented in UVM-SC versus UVM-SV back to back using an appropriate simulator is given.

II. Model driven software development

A. Our approach

Model driven software development can be implemented in many ways. Typically, a custom script which reads abstract specifications from a file is the weapon of choice. Based on experience and leveraging from more than a decade of development we have chosen to implement the UVM-XX generator in the open source Eclipse modeling Framework (EMF) [7].

B. EMF Features and Differences towards existing UVM generators

EMF provides some general benefits which existing UVM generators lack, like preservable user regions. This makes the user able to rerun the generator to automatically update the generated regions. Typically, all of the existing generators do not support this. Another great benefit is the easiness of updating the abstract specification, since model specifications can be specified in XSD, BNF and UML. Moreover, multiple input formats are independently supported. Lastly, the generator comes with an highly developed template backend for easy template adaption. The user will also get Eclipse context aware editing and syntax highlighting for free.

Figure 1 depicts the basic EMF flow where an abstract description of the model is provided and from that an EMF model is generated. The generated EMF model can then be filled with input and a template based backend can then generate the desired output.

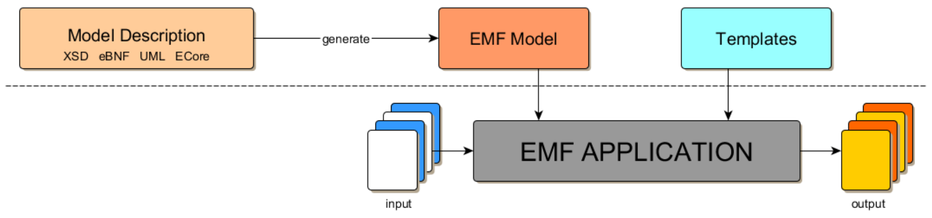

Figure 1: High level EMF flow description.

III. Abstracting UVM Test Benches

A. Pragmatism

Many examples have been given for abstracting an RTL testbench over time. Especially, UVM-based testbenches have been targeted during the last couple of years. Much pragmatism has to be applied when defining the abstraction to avoid making it too ambitious and thus very difficult to implement and to use. Looking at what major tasks a verification engineer is typically facing will bring the following to one's mind:

- **LUVC**: Implementation of UVCs.
- **LB2B**: Implementation of a back to back testbench for UVC verification.
- **LTB**: Implementation of an RTL testbench utilizing various UVCs. The UVCs can be generated but they can also be legacy Verification IP (VIP) etc.

These three major tasks are usually dependent on one another since mostly the verification engineers implement the UVCs, verify them and finally utilize them in the "real" world. Thus, a generator which supports the tasks via a layered abstract specification for each, would be beneficial.

One of the key points in using model driven software development is that the abstraction should save you from unnecessary code, meaning that it should compress information into a compact format. At the same time, the input format has to be easy to read. To address these two issues, we have chosen to define the abstraction as domain specific language (DSL) in which we can express all three layers and the relations between them. We call this DSL "Verification Environment format" (VE).

B. VE Format

EMF supports XTEXT [8] for defining an eBNF-like (Extended Backus–Naur Form) specification of the VE DSL. Thus, the EMF model for VE can be generated by specifying its grammar via XTEXT as shown in Figure 1. Selected parts of its definition are described related to each of the three layers mentioned in the section III.A.

As a single DSL is used for all three layers, a separation on syntax level is needed. This is achieved by categorizing them into two types: `VerificationComponent` (LUVC) or `EnvironmentComponent`

(LB2B, LTB). This is captured in XTEXT as shown in Figure 2. Note the `type` field which is either `vc` or `env` and the `mode` field which allows distinction between LB2B and LTB for the `EnvironmentComponent`.

Finding the correct abstraction level for specifying a UVC is the key. The main part is an abstraction for types and especially the driver/monitor code implementing the actual pin wiggling of the protocol should be abstracted as a part of the UVC abstraction. For the LUVC layer, section III.C describes directly how we handle types. For the protocol specific code, we have chosen to not abstract this for pragmatic reasons. As EMF supports preservable user regions we generate the driver, monitor etc. which contain protocol specific code without destroying it. The set of requirements for a LUVC abstraction boils down to:

- Being preferable confined to a separate scope for reusability (e.g. an SV package).
- Handle generic/reusable constants, types etc.
- Specify almost any hierarchical layout of the UVC (e.g. support multiple agents within a single UVC etc.) and different types of drivers (e.g. master/slave, producer/consumer etc.).
- Capture the RTL interface and the abstracted protocol transaction (UVM sequence item)..
- Support UVC configuration.

Figure 3 shows how all of these requirements are captured in the VE DSL as separate syntax blocks. Since the same pattern is used for all blocks and for brevity reasons then the remaining parts of this section only shows a few selected highlights of these blocks.

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

Figure 2: XTEXT VE DSL Specification of separation of LUVC, LB2B and LTB.

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

Figure 3: XTEXT VE DSL Specification of LUVC.

One of the most important blocks is the `InterfaceBlock` which captures the RTL interface of the protocol handled by the UVC. This part of the VE DSL, shown in Figure 4, captures the signals in the interface, their types, but also the constants/parameters, clock and reset etc. See section V.A for a concrete example.

```
InterfaceBlock:
        {InterfaceBlock} 'interface' '{'
        ( parameterBlocks += ParameterBlock
          | clockBlocks += ClockBlock
          | resetBlocks += ResetBlock
          | signalBlocks += SignalBlock)* '}'
;
ParameterBlock:
        {ParameterBlock} 'parameter' '{'
                parameterItems += ParameterItem* '}'
;
ParameterItem:
        ( 'name'  '=' name = STRING
          & 'type'  '=' type = STRING
          & 'value' '=' value = STRING
          & ('comment' '=' comment = STRING)? ) ';'
;
ClockBlock:
        {ClockBlock} 'clock' '{'
                clockItems += InterfaceSignal* '}'
;
```

```
ResetBlock:
        {ResetBlock} 'reset' '{'
                resetItems += InterfaceSignal* '}'
;
SignalBlock:
        {SignalBlock} 'signal' '{'
                signalItems += InterfaceSignal* '}'
;
InterfaceSignal:
        ( 'name'  '=' name = STRING
          & 'type'  '=' type = STRING
          & ('sc_type'  '=' sc_type = STRING)?
          & ('type_unpack'  '=' type_unpack = STRING)?
          & ('driver' '=' driver = STRING)?
          & ('resetval' '=' resetval = STRING)?
          & ('comment' '=' comment = STRING)?) ';'
;
```

Figure 4: XTEXT VE DSL Specification for the interface part of the UVC.

Another quite important part of a UVC is the specification of the abstract transaction which captures a transaction on the bus. Figure 5 shows how this is captured in the VE DSL in the `TransactionBlock` section which more or less lets the user specify abstract variables for capturing the protocol centric information but also meta information such as can it be randomized etc. See section V.A for an example.

The `EnvironmentComponent` defines the abstract information needed to generate testbenches for both the LB2B and LTB layers. The distinction between the two types of testbenches is needed since more code can be generated for LB2B testbenches as they are by nature simpler and only require a single generated UVC. An `EnvironmentComponent` can be generated by setting the `type` to `env` and specifying either `back2back` or `tb` in the `mode` field in order to obtain a back to back testbench or an RTL testbench, respectively. Additionally, the VE DSL allows multiple `VerifcompBlocks` for specifying which UVCs to instantiate, a `ScoreboardBlock` for expressing instantiation and hook up of the generic UVM scoreboard (see [9] for more details), a `EnvconfigBlock` for handling the configuration of the environment and finally multiple sequence and test blocks for specifying virtual sequences and tests to be used with the generated testbench. See Figure 6 for a VE DSL snippet and section V.B for an example. Note the support for importing other VE files.

```
TransactionBlock:
        {TransactionBlock} 'transaction' '{'
        ( dataBlocks += DataBlock
          | …
        '}'
;
DataBlock:
        {DataBlock} 'data' '{'
                dataItems += DataItem*
        '}'
;
DataItem:
        ( 'name'  '=' name = STRING
          & 'type'  '=' type = STRING
          & ('sc_type'  '=' sc_type = STRING)?
          & ('type_unpack'  '=' type_unpack = STRING)?
          & 'rand' '=' rand = STRING
          & ('default' '=' default = STRING)?
          & 'macro'  '=' macro = STRING
          & ('macroflag'  '=' macroflag = STRING)?
          & ('comment' '=' comment = STRING)?)
        ';'
;
```

Figure 5: XTEXT VE DSL Specification of an abstract transaction.

```
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
;
```

Figure 6: XTEXT VE DSL Specification of testbench blocks.

C. *Handling Types*

Types can be very difficult to handle and especially type conversion between two similar but not completely equivalent languages like SytemVerilog and SystemC. We even avoided implementing an abstraction for SV types in the UVM-SV only version of the generator simply to be pragmatic. We kept e.g. the type definition of a signal in the `InterfaceBlock` as a string for complete freedom. As we are extending the UVM-SV generator to support UVM-SC and its types we have three ways of implementing this:

1. Raising the abstraction level of types from a simple string to a real meta type, which should be a superset of at least the types in SystemC and SystemVerilog, to correctly generate types.
2. Implement a full SystemVerilog to SystemC type translator which would parse the existing SystemVerilog string and translate that into the appropriate SystemC type.
3. Implement a simpler version of (2) which has no parsing involved but simply some fixed mapping of the standard types and with the possibility to do user overwrites for complete freedom.

(1) requires a major rework of the current UVM-SV generator and the actual definition of the superset types. (2) could be embedded into the existing UVM-SV generator but requires a lot of code to handle all possible type translations. Hence, we have chosen (3) which actually doesn't break the option for a switch to (2) later on. Thus, the only thing we have had to change in the VE DSL is the addition of an optional `sc_type` string which can be seen in Figure 4 and Figure 5. Thus, types in the UVM-SV/SC generator will be resolved as follows:

- If a local `sc_type` has been specified, use that.
- If a user defined type translation has been specified for a certain SV type, then use that.
- If a default type translation has been specified for a certain SV type, then use that.
- If none of the 3 first types resolve rules do not apply, then signal SystemC type error.

IV. Generating UVM

Generating UVM source code is fairly straight forward due to the EMF framework. The general flow is depicted in Figure 7 which shows the EMF application UVMGen for each of the three layers. A set of UVM-SV and UVM-SC templates for each layer are traverses the VE model and produces UVM-SV/SC source code.

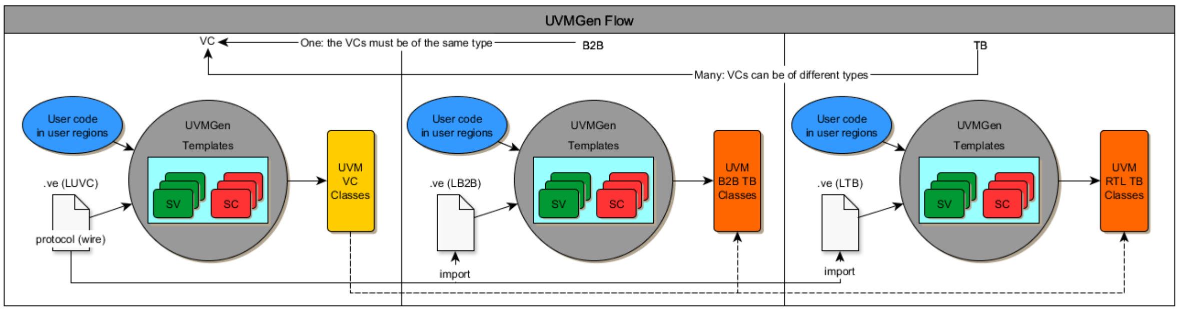

Figure 7: VE file input → VE EMF model → MTL Templates → UVM source code.

A. UVM – SystemVerilog (UVM-SV)

Invoking UVMGen with a LUVC, LB2B or LTB VE file and the UVM-SV templates will generate a set of UVM-SV classes. The classes implement quite standard UVM-SV except from a few exceptions:

- The virtual interfaces are distributed from toplevel→test→enviroment→UVCs to ensure maximum vertical reuse.
- The UVM config DB is deliberately not used very much since it embraces unorganized code.
- Parameters are distributed as class parameters for easy access.
- UVC components deliberately use implcit phasing (`run_phase`) to avoid phase jumping problems.
- UVM tests use explicit phasing for optimal testcase control.
- Virtual sequences are primarily implemented using the following design pattern: random member variables are created if needed in `pre_randomize()` and not in the `body()`. Thus, they are randomized when the sequence is randomized and cross constraints on them can be implemented. This can be cumbersome if all is handled in the `body()` task.

B. UVM – SystemC (UVM-SC)

Once we have UVM-SV generated then extension to also generate UVM-SC is quite easy since the EMF platform combined with the VE DSL gives the following advantages:

- Quite high abstraction within the VE DSL (One specification, Multiple implementations).
- Single EMF model with separates template sets (can handle the same abstraction individually in UVM-SV and UVM-SC).
- User regions for difficulty to generate parts.
- Elegant solution to the type conversion problem between SystemVerilog and SystemC.

V. Example

A brief walkthrough of the Wishbone example is hereby presented. The example utilizes the Wishbone protocol. Thus, a Wishbone UVC and a back-2-back testbench for this UVC have to be generated.

A. *Wishbone UVC*

For generating the Wishbone UVC, a VE file with `type` field equals to `vc` has been defined (Figure 8).

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

Figure 8: Initial VE code to generate a wishbone UVC.

The VE file is then filled up with the remaining blocks as described in the XTEXT file. Figure 9 shows the `agent`, `driver` and `interface` blocks: there is an instance of the agent, two drivers (`master` and `slave`), and the `interface` block containing definitions for the Wishbone interface.

```
//----------------------------------------------------------------------
// 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";   …  }
}
```

Figure 9: VE code for the Agent, Driver and Interface blocks.

```
//----------------------------------------------------------------------
// Wishbone protocol transaction definition
//----------------------------------------------------------------------
transaction {
  data {
    name="dat"          type="logic[WB_DATA_WIDTH-1:0]"     rand="yes"
      macro="int"       comment="Wishbone data";
    name="adr"          type="logic[WB_ADDR_WIDTH-1:0]"     rand="yes"
      macro="int"       comment="Wishbone address";
    name="sel"          type="logic[WB_DATA_BYTE_NUM-1:0]"  rand="yes"
      macro="int"       comment="Wishbone byte select";
    name="op"           type="tp_op"                        rand="yes"
      macro="enum"      comment="Wishbone operation";
    …  }
  delay {
    name="master_delay_cyc0_cyc1"   type="int"              rand="yes"
      macro="int"       macroflag="UVM_ALL_ON | UVM_NOCOMPARE"
      comment="Wishbone master delay before rising CYC";
    …  }
  delayconstraint {
    driver="master"      item="master_delay_cyc0_cyc1"
      nodelay_if="no_master_slave_delays"
      maxdelay="max_master_delay_cyc0_cyc1";   …  }
}
```

Figure 10: VE code for the Transaction block.

Figure 10 illustrates the `transaction` block: the transaction is composed by a set of data, delays and constraints to be applied on the delays. The data elements, which can be defined as random by setting the `rand` field, capture the different signals related to the Wishbone protocol (e.g. `dat` is the transferred data, `adr` the address, `op` defines if the transaction is a read or a write operation) and their type is specified by the `type` field.

Once the VE file is completed, it shall be passed as input to UVMGen which will then generate all the UVM classes for the Verification Component. Table I shows a fragment of the generated code for the UVM Agent class, with a comparison between UVM-SV and UVM-SC. Note the tags KEEP -> Start/End of user code delimiting the user regions.

Table I: Comparison of generated code for the UVC Agent: SystemVerilog (left) and SystemC (right)

```
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);
  }
};
```

B. *Wishbone back to back testbench*

The approach for generating the back to back testbench is very similar to the one applied for the generation of the Wishbone UVC, using a different VE file to obtain generated UVM classes. A VE file with `type` field equals to `env` has been defined (Figure 11).

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

Figure 11: Initial VE code to generate a wishbone B2B testbench.

Following the XTEXT syntax, the VE file is then completed with the remaining blocks. Figure 12 shows the `verifcomp` block, where it is defined which, how many and what variant (`master` or `slave`) verification components will be instantiated. The `envconfig`, `sequence`, and `test` blocks are then used for generating different tests which rely on different sequences and configuration files. The `base` field allows to define which file to extend from: in this case, with an empty `base` for all the sequences and tests, these will extend respectively from `cl_wb_b2b_base_seq` and `cl_wb_b2b_base_test`, which are generated by default.

```
//-----------------------------------------------------------------------
// 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

...
```

```
...
//-----------------------------------------------------------------------
// List of sequences to be used by tests
//-----------------------------------------------------------------------
sequence {
   name="basicseq"   base="";         // Declares cl_seq_basicseq,
extends cl_wb_b2b_base_seq
   name="randomseq"  base=""; }       // Declares cl_seq_randomseq,
extends cl_wb_b2b_base_seq
//-----------------------------------------------------------------------
// List of tests
//-----------------------------------------------------------------------
test {
   name="basictest"   base=""    topseq="basicseq"
envconfig="basicconf";
   name="randomtest"  base=""    topseq="randomseq"
envconfig="userconf"; }
```

Figure 12: VE code for Verifcomp, Envconfig, Sequence and Test blocks.

Once the VE file is completed, it shall be passed as input to UVMGen which will then generate all the UVM classes for the testbench. Table II shows a fragment of the generated code for the UVM Environment class, with a comparison between UVM-SV and UVM-SC. Note the tags `KEEP -> Start/End of user code` delimiting the user regions.

Table II: Comparison of generated code for B2B testbench environment: SystemVerilog (left) and SystemC (right).

```
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",
  uvm_config_db #(cl_wb_config)::set(this, "wb_monitor", "cfg",
this.cfg.wb_monitor_cfg);
  this.wb_monitor = cl_wb_env::type_id::create("wb_monitor", this);

// Connect environment
//---------------------------------------------------------------------
function void cl_wb_b2b_env::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  // Connect handles of local sequencers to virtual sequencer
  this.virtual_sequencer.wb_master_wishbone_agent_sequencer =
this.wb_master.wishbone_agent.sequencer;
  this.virtual_sequencer.wb_slave_wishbone_agent_sequencer =
this.wb_slave.wishbone_agent.sequencer;
```

```
cfg.wb_monitor_cfg);
    wb_monitor = cl_wb_env::type_id::create("wb_monitor", this);
  }
// Connect environment
//---------------------------------------------------------------------
  void connect_phase(uvm::uvm_phase& phase) {
    uvm::uvm_env::connect_phase(phase);

    // Connect handles of local sequencers to virtual sequencer
    virtual_sequencer->wb_master_wishbone_agent_sequencer =
      wb_master->wishbone_agent->sequencer;
    virtual_sequencer->wb_slave_wishbone_agent_sequencer =
      wb_slave->wishbone_agent->sequencer;
  }
};
```

VI. FUTURE EXTENSIONS

The current implementation only handles LUVC and LB2B for UVM-SC. It is obvious to extend this to handle LTB as well. Additionally, one can start looking into abstraction of functional coverage so that a common functional coverage model can be shared. This would of course require that something equivalent to SV functional coverage exists in the SC language. Additionally, UVMGen could be extended with many new features since there is no limit for how many template sets you can have. For instance, testbench documentation could be generated from the same EMF model including a figure showing the testbench architecture, since the EMF model would contain this information.

VII. CONCLUSION

We have proven that a pragmatic approach coupled with model driven software development techniques is a great universal solution for testbench generation. It is very easy to adapt to new UVM versions and the user regions are very useful to reuse this approach after the initial testbench generation which is a typical shortcoming of other generators. Furthermore, we defined an abstraction for specifying UVCs and UVM testbenches in general which can be extended. This abstract specification was demonstrated to be able to generate complex testbenches by showing side-by-side examples for UVM-SV and UVM-SC. A single user entry source (the abstract specification) was used to generate code for different target languages while being concrete enough to be useful. This fact and the nature of UVM, to separate the testbench from the DUT, makes it easy to reuse and gather tests over many designs to improve verification already on the earliest level. Reuse of testbenches on HiL-approaches for UVM-SC where shown in [5] which makes the methodology shown in this paper applicable in an even broader reuse aspect.

ACKNOWLEDGMENT

Special thanks to anyone involved in the development of UVMGen: Claus Rørbech, Kevin Steffensen, Igor Bosisio, Stefano Minigutti. Thank you for providing ideas, feedback etc.

REFERENCES

[1] Accellera Systems Initiative, Standard Universal Verification Methodology.

[2] IEEE Computer Society, 1800-2012 IEEE Standard SystemVerilog Language Reference Manual.

[3] IEEE Computer Society, 1666-2005 IEEE Standard SystemC Language Reference Manual.

[4] Wishbone SoC Interconnection Architecture, Revision B3, http://cdn.opencores.org/downloads/wbspec_b3.pdf .

[5] Paul Ehrlich, Thang Nguyen and Thilo Vörtler, "UVM-SystemC based hardware in the loop simulations for accelerated Co-Verification", Design and Verification Conference (DVCon) Europe, 2014.

[6] OpenCores, http://www.opencores.org.

[7] Eclipse Modeling Framework (EMF), https://eclipse.org/modeling/emf .

[8] XTEXT Framework, https://eclipse.org/Xtext/ .

[9] Jacob Sander Andersen, Peter Jensen, Kevin Steffensen, "Versatile UVM Scoreboarding", Design and Verification Conference (DVCon) Europe, 2014.