Using Synopsys VCS to connect a Company's SystemC Verification Methodology to Standard Concepts of UVM

Frank Poppen
OFFIS Institute for Information Technology
Tel.: +49(441)9722-230
frank.poppen@offis.de

Marco Trunzer
Robert Bosch GmbH
Tel. +49(7121)35-2981
Marco.Trunzer@de.bosch.com

Jan-Hendrik Oetjens
Robert Bosch GmbH
Tel.: +49(7121)35-4684
Jan-Hendrik.Oetjens@de.bosch.com

Over the last decades, intelligent electronics in heterogeneous systems improved all aspects of everyone's daily life. An advantage a modern civilization cannot ignore. The increasing complexity of the electronic components though, makes us dependent on solving a growing design verification challenge. Especially knowing, that safety relevant functionality as in automotive driving is part of this development. Standardized as well as proprietary concepts, languages and tools line up for the task [6]. Unfortunately, there is no such thing as one size fits all in this. Verification engineers need to choose and combine what fits best for the company, the design-team and application domain. They create company's verification strategies with deep roots into the design process. Changes to the strategy need to be done carefully and incrementally to ensure continued productivity.
This work is a twin of the previously published paper [1] and a copy of that in big parts enhanced by more code examples in the Appendixes A to C. In [1] we compared concepts of UVM [9] and showed how UVM components are instantiable in our SystemC (SC) based IFS [3] test environment using UVM Connect to verify designs specified in VHDL (-AMS), SystemC (-AMS), Verilog (-AMS) or Matlab/Simulink [4]. There we demonstrated that our approach and UVMC do not depend on proprietary technology and presented the applicability for mixed language simulators from Mentor and Cadence. This paper is to extend the claim to cover Synopsys vcs-mx as another mixed language simulation environment: we make use of SystemVerilog UVM inside a SystemC test bench connected with UVMC using Synopsys' simulator vcs-mx.

1. Introduction

Automotive electronic today shows a constant increase in performance, number of subsystems and their interoperability to compose most complex heterogeneous systems. Guaranteeing properties of safety, sustainability and comfort is a challenge, which requires consistent verification of quality along every stage of the development process. At the level of integrated circuit design EDA industry promotes the Universal Verification Methodology (UVM) [9]. Such relatively new standards need to find their way into, often in-house, verification concepts that have been out long before UVM and SystemVerilog (SV). Those concepts distinguish themselves from standard concepts in that they are more tailored to relevant use scenarios and efficient usage in special contexts. Unfortunately, when using non-standard verification methodologies, one cannot access and profit from the vast amount of resources available like third party verification Intellectual Property (IP) and skilled human resources. On the other hand switching completely to UVM as a standard method means dispensing the already available in-house verification IP and the benefits of a tailored solution. Because of that even when there are good reasons to stay with an in-house verification methodology, it becomes necessary to interface to standard methodologies. So like already stated in [2], an evolution of the verification method is more desirable than a radical revolution.
The methodology named IFS ("Integrated Functional verification Script environment") was continuously enhanced from VHDL with VHDL-AMS [2] to SC [3], Matlab/Simulink [4] and

now further on to SV and UVM. Major aspects of the test bench architecture as defined by IFS can also be found in the established standard UVM that has its roots in the Open Verification Methodology (OVM) and the Verification Methodology Manual (VMM). Figure 1 matches basic concepts of both approaches that are directly comparable. With removed details of UVM's concepts, the IFS approach is less complex and simple to apply. Moreover, with VHDL AMS the IFS methodology already includes analog/mixed signal (AMS) simulation at no additional cost including constraint-random capabilities. After an afternoon introduction (analog) designers and system integrators are able to use test benches and create command files for own test cases. All stakeholders in the development process, digital designer, analog designer, verification engineer and system engineer, make use of the same, simple IFS command language to create (self-checking) test cases.

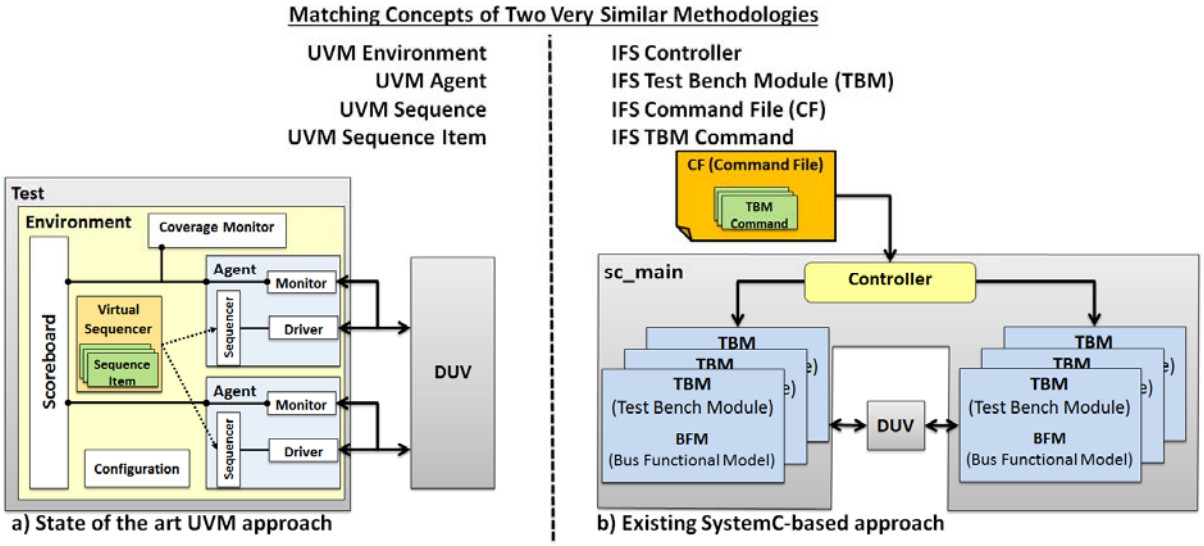

Figure 1: Juxtaposition of a) UVM and b) IFS.

We utilized a bus arbiter test case (Chapter 2) as design under verification (DUV) inside an IFS simulation environment with bus master and bus slave Test Bench Modules (TBM) (Chapter 3). In the architecture of our experiment (Chapter 4) bus masters were successfully replaced by UVM agents and fully simulated in a holistic test bench simulation using Synopsys VCS (Chapter 5). The document concludes in Chapter 6. The listing of Appendix A shows the source code of our TBM master wrapper which handles all three proprietary foreign language instantiation concepts of the three simulators Synopsys VCS, Mentor Questa Simulator and Cadence Incisive. Appendix B contains the compile and run script used to execute the SystemC-UVMC-SystemVerilog mixed language simulation example.

2. Bus Arbiter Example as Test Case

We chose a switched bus arbiter implementation as a test case for this work. It connects a configurable number of bus masters with a configurable number of memory bus slaves for read or write accesses (refer to Figure 2). The design is simple enough to be understandable within short time and allows quick implementation with little risk of introducing bugs. We used it for several mixed-language simulations evaluations in the past and it is now available in VHDL, Verilog and SC together with matching models of masters, slaves and test bench setups. In the scope of this work, we completed this collection with a UVM/SV test environment.

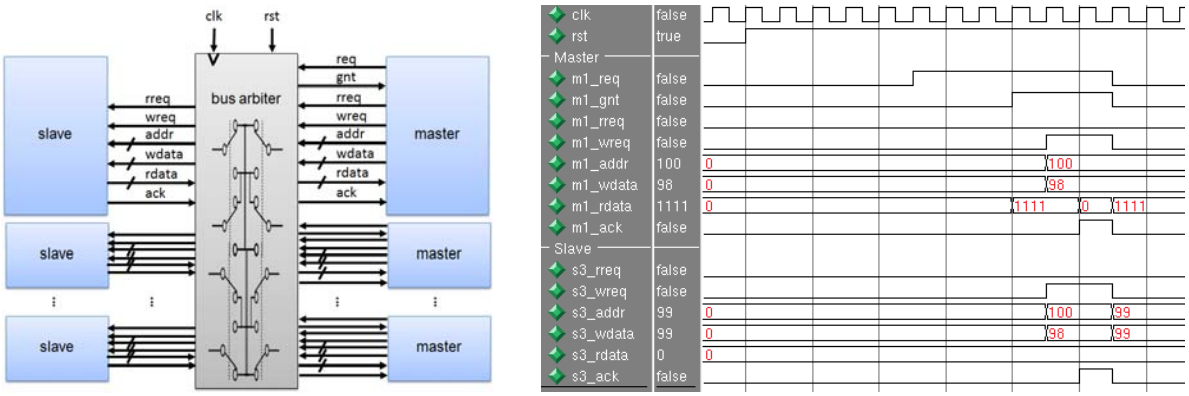

Figure 2: Switched bus arbiter example as DUV for the experiment of this work.

3. Integrated Functional Verification Script Environment and Bus-Arbiter Example

A first introduction into Integrated Functional Verification Script Environment (IFS) is to be found in [2] and [3]. IFS is tailored to the domain of automotive electronics system design and specifically satisfies the needs of engineers and system integrators for efficient test implementation. IFS is a SC-based library that compiles and simulates with any simulator complying with the IEEE 1666 SystemC standard. Figure 3 depicts the setup of the bus arbiter example in an IFS environment.

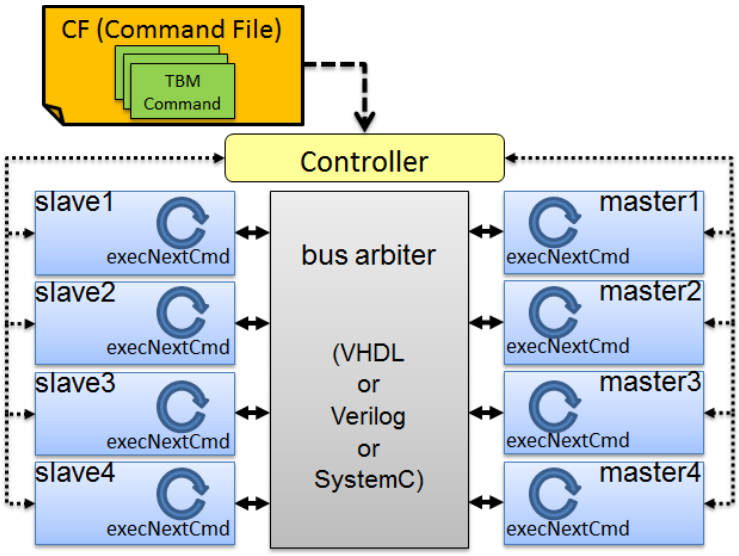

Figure 3: IFS simulation environment for a bus arbiter test bench.

In a mixed-language simulation environment the DUV's implementation could be any language like VHDL, Verilog, or SystemC, as long as the DUV's interfaces are hooked up to IFS Test Bench Modules (TBM). A TBM acts as Bus Functional Model (BFM) and generates stimuli for the DUV. All TBMs are derived from the class `IFS_ModuleBase` which itself is derived from `sc_module`. Therefore, a TBM is a SC module enriched by predefined IFS commands (`print <text>`, `wait <time|event>`, `sync <TBMlist>`, `reportlevel <severity>`, `notify <event>`, `assign <envVar>`, `quit`, and other). TBM developers need to implement the BFM part of the behavior by user-defined commands and register them with the IFS controller at runtime. User defined commands are callable from the command file (CF) the same way as predefined commands. Scheduled in parallel by the simulation kernel, the TBMs each execute an endless loop in which they request the next command from the IFS controller's CF. Execution is suspended on wait or sync. TBMs terminate on reaching the last command or if the quit command is issued explicitly. The concept allows the quick modification of test runs by exchanging or modifying

the CF. SC code of test bench and TBM remains unchanged between different test runs removing recompilation and re-elaboration from the verification process.
Additionally, TBMs in VHDL and VHDL-AMS are supported. They are automatically connected to the SC IFS controller by no more than specifying a unique TBM command name (for CF) and a unique TBM ID number. It is even possible to describe the test bench structure and interconnection in any HDL by using the standard mixed-language capabilities of the simulators.
The left listing demonstrates a simple CF using predefined commands, as well as user defined TBM commands (`Set_Offset`, `Set_I_Wait`, `Write` and `Read`) of four bus masters accessing four memory slave modules through the DUV. The right shows a simulation run with OSCI reference simulator.

```
CLK PERIOD 10 ns
CLK RESET  0  12

ALL SYNC ALL

SL1 print "Config slave 1!"
SL1 Set_Offset 300
SL1 Set_I_Wait 100
SL2 print "Config slave 2!"
SL2 Set_Offset 200
SL2 Set_I_Wait 99
SL3 print "Config slave 3!"
SL3 Set_Offset 100
SL3 Set_I_Wait 47
SL4 print "Config slave 4!"
SL4 Set_Offset 0
SL4 Set_I_Wait 69

ALL SYNC ALL

#loop 4
  ALL SYNC ALL
  MS1 Write $(100*#i) $(100+#i)
  MS1 Read  $(100*#i) $(100+#i)
#eol

ALL SYNC ALL

SL1 print "End Of Test Script"

ALL QUIT
```

```
  SystemC 2.3.0-ASI --- Nov 29 2013 14:57:17
  Copyright (c) 1996-2012 by all Contributors,
  ALL RIGHTS RESERVED
INFO (0 s) Loading script: 'control.cmd'
INFO (0 s) Finished loading
[CLK,0 s] activate system clock of 10 ns
[CLK,0 s] reset gets active
[CLK,120 ns] reset gets passiv
INFO (120 ns)[SL1] Config slave 1!
[SL1,120 ns] set slave_offset = 300
[SL1,120 ns] set int_wait_cycles = 100
INFO (120 ns)[SL2] Config slave 2!
[SL2,120 ns] set slave_offset = 200
[SL2,120 ns] set int_wait_cycles = 99
INFO (120 ns)[SL3] Config slave 3!
[SL3,120 ns] set slave_offset = 100
[SL3,120 ns] set int_wait_cycles = 47
INFO (120 ns)[SL4] Config slave 4!
[SL4,120 ns] set slave_offset = 0
[SL4,120 ns] set int_wait_cycles = 69
[MS1,130 ns]     Write    (Address: 0, Value: 100)
[SL4,140 ns]     Write    (Address: 0, Value: 100)
[SL4,210 ns]     Read     (Address: 0, Value: 100)
[MS1,220 ns]     Read     (Address: 0, Value: 100)
[MS1,270 ns]     Write    (Address: 100, Value: 101)
[SL3,280 ns]     Write    (Address: 100, Value: 101)
[SL3,350 ns]     Read     (Address: 100, Value: 101)
[MS1,360 ns]     Read     (Address: 100, Value: 101)
[MS1,410 ns]     Write    (Address: 200, Value: 102)
[SL2,420 ns]     Write    (Address: 200, Value: 102)
[SL2,490 ns]     Read     (Address: 200, Value: 102)
[MS1,500 ns]     Read     (Address: 200, Value: 102)
[MS1,550 ns]     Write    (Address: 300, Value: 103)
[SL1,560 ns]     Write    (Address: 300, Value: 103)
[SL1,630 ns]     Read     (Address: 300, Value: 103)
[MS1,640 ns]     Read     (Address: 300, Value: 103)
INFO (640 ns)[SL1] End Of Test Script
INFO (640 ns) SIMULATION END FROM COMMAND FILE
INFO (640 ns) Exiting simulation.
Info: /OSCI/SystemC: Simulation stopped by user.
INFO (640 ns) Report:
INFO (640 ns) Encountered errors: 0
INFO (640 ns) Encountered warnings: 0
```

4. Architecture of the Experiment

Even though TLM and UVM concepts are not bound to a certain design language per se, they are practically not available in all flavors languages. SV is the choice of implementation for UVM. SC is yet missing out on UVM concepts, but literature shows work in progress [7]. The similarities of the IFS and UVM concepts (Figure 1) suggested a common basis for an interchangeable use. To proof the assumption in an experiment, we needed to connect SC/IFS and SV/UVM. The Verification Academy offers UVM Connect to interconnect the two: “UVMC is an open-source UVM/OVM-based library that provides TLM1 and TLM2 connectivity and object passing between SC and SV UVM/OVM models and components. It also provides a UVM Command API for accessing and controlling UVM simulation from SC (or C or C++). UVM Connect allows you to reuse your SC architectural models as reference models in UVM/OVM verification and/or reuse SV UVM/OVM agents to verify models in SC.” [8]

We demonstrate here that the UVMC API is applicable in the substitution of an IFS TBM by an UVM agent in a mixed-language simulation that combines SC including SCV (SystemC Verification Standard) and SV/UVM. This way we open the door to make a full evolutionary inclusion of state of the art UVM verification IP in the well-established IFS flow including its link to AMS simulation.

A TBM directly correlates to the functionality of an UVM agent. Both act as transactor between test bench and DUV and translate messages/commands to bit wiggles. For complex IP it can become quite cumbersome to create a verified TBM. When IP comes with a UVM test environment, TBM reimplementation is redundant effort, if we could reuse the delivered UVM agents instead. In the remainder of this chapter, we show how this is achieved. The presented concept does not rely on the IFS library. It is generally applicable in SC.

UVM Connect (UVMC) makes use of the SV Direct Programming Interface (DPI) and enables the communication of a SC model with a UVM model. Both language models are compiled separately and co-exist in parallel in a mixed-language simulator environment. It is possible to exchange TLM messages between the two, as well as exchange control commands (compare with Figure 4). This is the intended use of UVMC between SC and SV.

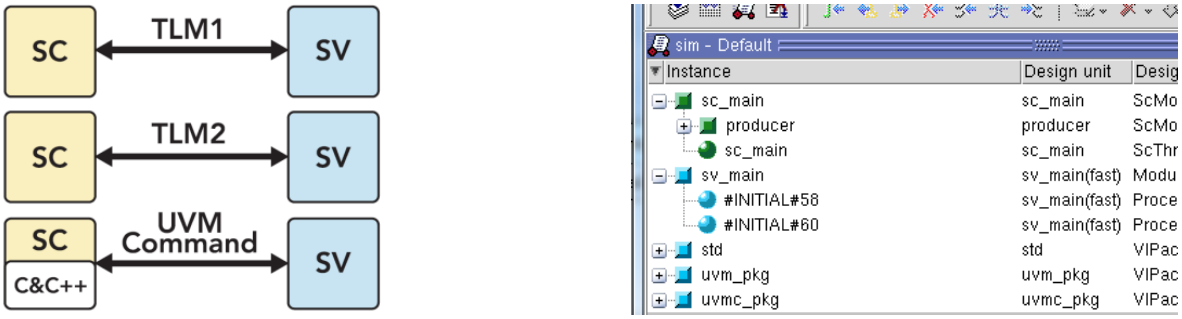

Figure 4: UVMC as link for TLM and commands between SC and SV (source: UVMC documentation).

Unfortunately, this is not ideal for the use case of this work. We need a standalone UVM agent without UVM environment to replace a TBM inside SC. The abstraction level of the DUV model is register transfer level (RTL) and does not implement TLM ports. The agent should communicate to the DUV via its UVM driver at signal level, and not via TLM. We therefore use the option to instantiate a foreign language module from a SC wrapper instead. The methodology is described in the mixed language simulators' user guide ([10], [11] and [12]) and has two advantages. Firstly, the agent's driver receives full pin level access to the DUV's interface. Secondly, the test bench architecture is defined in the SC source code only. If we change the test bench configuration of the bus arbiter for more masters, only the SC `sc_main` needs to be modified to hook up additional wrapper modules. Each wrapper instantiates the required UVM code by itself. The verification engineer does not need to touch or even know UVM/SV source code.

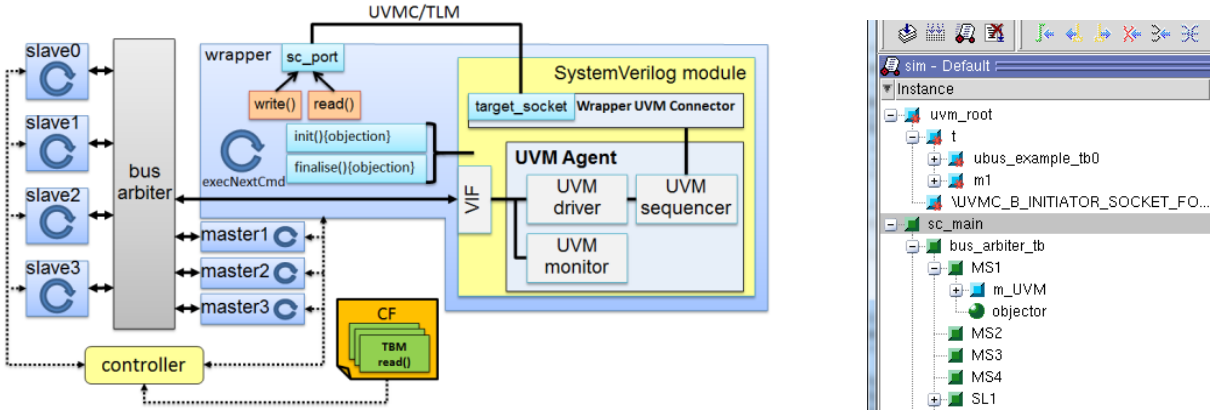

Figure 5: SC instantiating UVM in wrapper module using the foreign module interface.

Right side of Figure 5 shows a screenshot of an UVM agent instantiated inside SC using the `elaborate_foreign_module(hdl_name)` functionality of Mentor Questa. SC and UVM are interweaved with bare signal access between the two. The left side of Figure 5 shows the implemented architecture of the experiment. A wrapper TBM derived from the class `IFS_ModuleBase` acts just like any other TBM with predefined commands and next command execution loop, but fulfills three additional objectives. Firstly, the UVM agent is instantiated in the constructor of the wrapper (yellow rectangle). Instantiation implementation is proprietary to the chosen multi-language simulation environment. We evaluated Synopsys' VCS (`HDL_MODULE [10]`), Mentor Graphics' Questa simulator (`sc_foreign_module [11]`) and Cadence's Incisive (`ncsc_foreign_module [12]`). The concepts of the three simulators a very similar but still need individual coding. Our source code for this is listed in Appendix A.
Secondly, the wrapper implements a method to utilize UVMC's command API to set an objection for the UVM run phase. The objection is mandatory as otherwise the UVM simulation of the single UVM agent would terminate right after start because of missing sequences and sequence items. Thirdly, the wrapper implements commands that are twins to the UVM agent's processable sequence items. The following is a brief outline on the modus operandi:

- Start simulator, elaborate SC and SV is instantiated inside SC.
- `run 0` starts execution of the SV phases build, connect and the run phase.
- SC wrapper sets objection for the run phase. SV will not leave run phase until removed.
- `run all` executes all TBMs' execute-next-command loops. Wrapper receives user defined command call from IFS controller's CF.
- Wrapper calls implementation method of user defined IFS command. Method creates and sends TLM message via UVMC.
- Wrapper UVM connector is a UVM component derived from the UVM monitor class. It receives the TLM message and creates an according sequence item. The sequence item is forwarded through the UVM sequencer to the UVM driver.
- UVM Driver applies signal events to DUV's interface. IFS command finished execution.
- Execute-next-command repeats until quit or last command is reached. Wrapper removes UVM run-phase objection by call to wrapper's finalize method. Simulation ends.

The agent's original code for sequencer, driver and monitor remains unchanged. The IFS wrapper is fully reusable for future use in different environment configurations. It is fully transparent to the IFS test developer who will not get into contact with UVM code behind the TBM wrapper.

5. Report of Simulation Run

We evaluated our approach with three tool setups. The first setup consists of Mentor Graphics Questa 10.1c, UVM 1.1b and UVMC v2.2. The second setup utilized Cadence Incisive 13.10-s005 and the third bases on Synopsy VCS-MX vJ-2014.12-1. We believe that any setup complying with IEEE 1666 SC and IEEE 1800 SV should work.
The following is a full simulation run of SC/IFS instantiating a SV/UVM agent inside a TBM wrapper and connecting the two using UVMC. We removed repeating lines, blank lines and lines with little information to shorten the trace. Lines 1 to 10 document the elaboration phase.

```
1:  ********** wrap_uvm_master::CTOR(): Connecting TLM port
2:  Connecting an SC-side proxy chan for 'bus_arbiter_tb.MS1.port_10' with lookup string
    'sc_wrap_MS1' for later connection with SV
3:  ********** uvm_test::CTOR(): instantiating UVMC test
4:  INFO (0 s)[bus_arbiter_tb.MS1] Registered module 'bus_arbiter_tb.MS1'
5:  ...(also MS2 to MS4)
6:  INFO (0 s)[bus_arbiter_tb.SL1] Registered module 'bus_arbiter_tb.SL1'
7:  ...(also SL2 to SL4)
8:  INFO (0 s)[bus_arbiter_tb.CLK] Registered module 'bus_arbiter_tb.CLK'
9:  INFO (0 s) Loading script: 'control.cmd'
10: INFO (0 s) Finished loading
```

Run of simulation starts in line 11, where UVM begins to traverse through its simulation phases build, connect and finally entering the main run phase with line 28. Connecting the TLM ports between SC and UVM is established in two steps. Firstly, ports are created and labeled with a lookup string (SC line 2 above, UVM line 26 below).

```
11: Chronologic VCS simulator copyright 1991-2014
12: Contains Synopsys proprietary information.
13: Compiler version J-2014.12-1; Runtime version J-2014.12-1;  Feb 20 11:15 2015
14: ----------------------------------------------------------------
15: UVM-1.1d.Synopsys
16: (C) 2007-2013 Mentor Graphics Corporation
17: (C) 2007-2013 Cadence Design Systems, Inc.
18: (C) 2006-2013 Synopsys, Inc.
19: (C) 2011-2013 Cypress Semiconductor Corp.
20: ----------------------------------------------------------------
21: ********** wrap_uvm_master::initialiseModule(): Raising objection for UVM phase 'run'!
22: ----------------------------------------------------------------
23: UVMC-2.2
24: (C) 2009-2012 Mentor Graphics Corporation
25: ----------------------------------------------------------------
26: Registering SV-side 'sc_wrap_MS1.ifs_monitor.in' and lookup string 'sc_wrap_MS1' for later
    connection with SC
27: UVM_INFO @ 0 ns: reporter [RNTST] Running test ...
```

The wrapper raises an objection for the UVM run phase (line 21). UVMC connects the SC and UVM TLM ports by referring to their label (line 28). In line 34 the execute-next-command-loop receives a `MS1 Write 100 98` user-defined command from the IFS controller's CF and the wrapper's write method creates and sends a proper TLM message. In line 35, the wrapper UVM connector receives the TLM message and creates a sequence item 36. The UVM agent's driver consumes the sequence item and applies according signal events to the DUV (line 37). The SC slave TBM responds to the signal events (line 38).

```
28: Connected SC-side 'bus_arbiter_tb.MS1.port_10' to SV-side 'sc_wrap_MS1.ifs_monitor.in'
29: UVM_INFO ../tb_uvm/ubus_example_master_seq_pkg.sv(134) @ 0 ns:
    sc_wrap_MS1.sequencer@@master_memory_seq [master_memory_seq] master_memory_seq starting...
30: [bus_arbiter_tb.CLK,100 ns] reset gets passiv
31: [bus_arbiter_tb.SL1,100 ns] set slave_offset = 300
32: ... (configuring SL1 to SL4)
33: UVM_INFO ../tb_uvm/ubus_master_driver_pkg.sv(89) @ 110 ns: sc_wrap_MS1.driver
    [ubus_master_driver] ********* ubus_master_driver::get_and_drive(): Waiting for Item on
    seq_item_port!
34: ********** wrap_uvm_master::Write(): Sending payload '{cmd:2 parameters:100 98} to MS1.socket
    at time 150
35: UVM_INFO ../tb_uvm/ifs_command_monitor_pkg.sv(75) @ 150 ns: sc_wrap_MS1.ifs_monitor
    [ifs_command_monitor] ********* ifs_command_monitor::b_transport(): SC-TLM communication
    received: cmd - 00000002, parameters - '{"100", "98"}
36: UVM_INFO ../tb_uvm/ifs_command_monitor_pkg.sv(116) @ 150 ns: sc_wrap_MS1.ifs_monitor
    [ifs_command_monitor] ********* ifs_command_monitor::peek(): Informing driver to drive cmd-
    2, addr- 100, data- 98!
37: UVM_INFO ../tb_uvm/ubus_master_driver_pkg.sv(91) @ 150 ns: sc_wrap_MS1.driver
    [ubus_master_driver] ********* ubus_master_driver::get_and_drive(): Received Item on
    seq_item_port!
38: [bus_arbiter_tb.SL3,190 ns] Write (Address: 100, Value: 98)
39: UVM_INFO ../tb_uvm/ubus_master_monitor_pkg.sv(173) @ 195 ns: sc_wrap_MS1.monitor
    [ubus_master_monitor] collect_data_phase.
```

The UVM agent's monitor listens on the signal events between UVM agent and DUV, and recognizes a correct write transaction (lines 40 to 50).

```
40:  UVM_INFO    ../tb_uvm/ubus_master_monitor_pkg.sv(137)    @    195    ns:    sc_wrap_MS1.monitor
     [sc_wrap_MS1.monitor] Master transfer collected :
41:  ----------------------------------------------------------
42:  ubus_transfer_inst  ubus_transfer         -     @6690
43:    read_write        ubus_read_write_enum  32    WRITE
44:    addr              integral              32    'h64
45:    data              integral              32    'h62
46:    master            string                11    sc_wrap_MS1
47:    slave             string                0     ""
48:    begin_time        time                  64    180 ns
49:    end_time          time                  64    195 ns
50:  ----------------------------------------------------------
```

Lines 51 to 54 demonstrate that the other TBM masters operate in parallel just as well issuing write commands that are answered by the TBM slaves.

```
51:  [bus_arbiter_tb.MS2,820 ns]       Write    (Address: 16, Value: 32)
52:  [bus_arbiter_tb.SL4,830 ns]       Write    (Address: 16, Value: 32)
53:  [bus_arbiter_tb.MS4,1010 ns]      Write    (Address: 100, Value: 98)
54:  [bus_arbiter_tb.SL3,1020 ns]      Write    (Address: 100, Value: 98)
```

On reaching the `quit` command the run-phase objection is being dropped (line 60) and the simulation terminates.

```
55:  INFO (2050 ns)[bus_arbiter_tb.SL1] ---------------------------------------------
56:  INFO (2050 ns)[bus_arbiter_tb.SL1] End Of Test Script Reached...
57:  INFO (2050 ns)[bus_arbiter_tb.SL1] ---------------------------------------------
58:  INFO (2050 ns) Exiting simulation.
59:  SystemC: simulation stopped by user.
60:  ************ wrap_uvm_master::finaliseModule(): Dropping objection for UVM phase 'run'!
61:             V C S   S i m u l a t i o n   R e p o r t
62:  Time: 2050000 ps
```

6. Conclusions

We show that it is possible to instantiate an UVM agent inside a SC test bench and control it to generate signal events for a DUV in a SC test environment. Our objective was to reuse external UVM verification IP in our SC-based IFS test environment. The introduced approach is independent from the IFS library and therefore generic and applicable for any SC-based test environment. We have to admit though, that the complex interference between SV, UVM, UVMC, SC and IFS, opens a wide potential for unclear behavior and bugs. Instantiating single UVM components outside a UVM environment and using UVMC on top is not the intended use case by design of these technologies. Documentation in this corner use case is slim and sometimes trial and error was the only solution to solve unclear error messages.

There is hope from the charter of the Accellera Multi Language Working Group (MLWG) "to create a standard and functional reference for interoperability of multi-language verification environments and components." The MLWG realizes that VIP integration and interoperability problems are encountered frequently and not only between SC and SV as also stated in [6]. Standardization is mandatory and should contribute to our future work on this topic.

The result of this work is practically applicable. The standards for SV/DPI and SC are mature enough to issue no portability problems between the three simulation environments we used. For future work, it is of interest to turn our approach upside down and instantiate an IFS TBM as UVM agent inside an UVM test bench. From the gained experience, we believe this to be a feasible approach.

Acknowledgements: This work has been funded by the German Federal Ministry for Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under the grant 01IS13022 (project EffektiV). The content of this publication lies within the responsibility of the authors.

7. Literature

[1] F. Poppen, M. Trunzer, J.-H. Oetjens, “Connecting a Company’s Verification Methodology to Standard Concepts of UVM”, DVCon Europe, 2014.

[2] P. Jores, P. Borthen, R. Dölling, H.-W. Groth, T. Halfmann, S. Kern, M. Lampp, M. Olbrich, M. Pfost, R. Popp, D. Pronath, P. Rotter, S. Steinhorst, G. Wachutka, Y. Wang and S. Weber, “Verifikation analoger Schaltungen (Kurztitel: VeronA)”, Schlussbericht zur BMBF-Förderinitiative IKT2020, 2009.

[3] R. Lissel and J. Gerlach, “Introducing new verification methods into a company's design flow: an industrial user's point of view”, DATE 2007.

[4] K. Hylla, J.-H. Oetjens and W. Nebel, “Using SystemC for an Extended MATLAB/Simulink Verification Flow”, FDL 2008.

[5] R. Görgen, H. Kleen, J.-H. Oetjens, P. Jores and W. Nebel, “SystemC Based Verification of Complex Heterogeneous Systems”, Cyber-Physical Systems – Enabling Multi-Nature Systems, CPMNS 2012.

[6] J.-H. Oetjens, N. Bannow, M. Becker, O. Bringmann, A. Burger, M. Chaari, S. Chakraborty, R. Drechsler, W. Ecker, K. Grüttner, T. Kruse, C. Kuznik, H. M. Le, A. Mauderer, W. Müller, D. Müller-Gritschneder, F. Poppen, H. Post, S. Reiter, W. Rosenstiel, S. Roth, U. Schlichtmann, A. von Schwerin, B.-A. Tabacaru and A. Viehl, “Safety Evaluation of Automotive Electronics Using Virtual Prototypes: State of the Art and Research Challenges”, Proceedings of the 51th Design Automation Conference (DAC) 2014, San Francisco, CA, USA.

[7] M. Barnasconi, F. Pecheux and T. Vörtler, “Advancing System-Level Verification using UVM in SystemC”, Design and Verification Conference, DVCon 2014.

[8] Adam Erickson, “Introducing UVM Connect”, at https://verificationacademy.com/sessions/introduction-uvm-connect.

[9] Accellera, “Universal Verification Methodology (UVM) 1.1 User’s Guide”, May 18, 2011.

[10] Synopsys, “VCS® MX/VCS® MXi™ User Guide”, J-2014.12, December 2014.

[11] Mentor, “Questa® SIM User’s Manual Including Support for Questa SV/AFV”, 2012.

[12] Cadence, “SystemC Simulation User Guide”, 2012.

8. Appendix A

The listing of Appendix A shows the source code of our TBM master wrapper which handles all three proprietary foreign language instantiation concepts of the three simulators Synopsys VCS (`#define VCS`), Mentor Questa Simulator (`#define MTI_SYSTEMC`) and Cadence Incisive (`#define NC_SYSTEMC`).

Manually coded:

```
#if defined MTI_SYSTEMC
#  ifndef NC_SYSTEMC
#     ifndef VCS
      class foreign_module_master : public sc_foreign_module
      {
          public:
          sc_in_clk     uvm_master_clk;
          sc_in<bool>   uvm_master_rst;   // reset
          sc_out<bool>  uvm_master_req;   // master request
          sc_in<bool>   uvm_master_gnt;   // grant from arbiter
          sc_out<bool>  uvm_master_rreq;  // read request
          sc_out<bool>  uvm_master_wreq;  // write request
          sc_out<int>   uvm_master_addr;  // address
          sc_out<int>   uvm_master_wdata; // write data
          sc_in<int>    uvm_master_rdata; // read data
          sc_in<bool>   uvm_master_ack;   // acknowledge

          foreign_module_master(sc_module_name nm, const char* hdl_name)
           : sc_foreign_module(nm),
             uvm_master_clk("clk"),
             uvm_master_rst("rst"),
             uvm_master_req("req"),
             uvm_master_gnt("gnt"),
             uvm_master_rreq("rreq"),
             uvm_master_wreq("wreq"),
             uvm_master_addr("addr"),
             uvm_master_wdata("wdata"),
             uvm_master_rdata("rdata"),
             uvm_master_ack("ack")
          {
              cout  <<  "*************  foreign_module_master::CTOR():  Elaborating  foreign  module:  "  <<
hdl_name << endl;
              // param1: module name to take from worklib
              // param2: number verilog module parameters
              // param3: UVM name of module and TLM-Port identifier
              std::string tmp = "str_param=\"";
              tmp += nm;
              tmp += "\"";
              const char* generic_list[1];
              generic_list[0] = strdup(tmp.c_str());
              elaborate_foreign_module(hdl_name, 1, generic_list);
          }
          ~foreign_module_master()
          {}
      };
#     else
#        error "Cannot compile for MTI_SYSTEMC and VCS simultaniously!"
#     endif
#  else
#     error "Cannot compile for MTI_SYSTEMC and NC_SYSTEMC simultaniously!"
#  endif
#elif defined NC_SYSTEMC
#  ifndef MTI_SYSTEMC
#     ifndef VCS
          class foreign_module_master : public ncsc_foreign_module
          {
              public:
              sc_in_clk     uvm_master_clk;
              sc_in<bool>   uvm_master_rst;   // reset
              sc_out<bool>  uvm_master_req;   // master request
              sc_in<bool>   uvm_master_gnt;   // grant from arbiter
              sc_out<bool>  uvm_master_rreq;  // read request
              sc_out<bool>  uvm_master_wreq;  // write request
              sc_out<int>   uvm_master_addr;  // address
              sc_out<int>   uvm_master_wdata; // write data
              sc_in<int>    uvm_master_rdata; // read data
              sc_in<bool>   uvm_master_ack;   // acknowledge

              foreign_module_master(sc_module_name nm, const char* hdl_name)
               : ncsc_foreign_module(nm),
                 str_param(nm),
                 hdl_instance(hdl_name),
                 uvm_master_clk("clk"),
                 uvm_master_rst("rst"),
                 uvm_master_req("req"),
                 uvm_master_gnt("gnt"),
                 uvm_master_rreq("rreq"),
                 uvm_master_wreq("wreq"),
```

```
                uvm_master_addr("addr"),
                uvm_master_wdata("wdata"),
                uvm_master_rdata("rdata"),
                uvm_master_ack("ack")
            {
                cout << "************* foreign_module_master::CTOR(): Elaborating foreign module: " <<
hdl_instance << endl;
                ncsc_set_hdl_param("str_param", str_param);
            }
            const char* hdl_name() const { return hdl_instance; }

            ~foreign_module_master() {}

            private:
            std::string str_param;
            const char* hdl_instance;
        };
#     else
#         error "Cannot compile for NC_SYSTEMC and VCS simultaniously!"
#     endif
#   else
#     error "Cannot compile for MTI_SYSTEMC and NC_SYSTEMC simultaniously!"
#   endif
#elif defined VCS
#   ifndef MTI_SYSTEMC
#     ifndef NC_SYSTEMC
        #include "csrc/sysc/include/sv_uvm_master.h"
        class foreign_module_master {
            public:
            sc_in_clk     uvm_master_clk;
            sc_in<bool>   uvm_master_rst;   // reset
            sc_out<bool>  uvm_master_req;   // master request
            sc_in<bool>   uvm_master_gnt;   // grant from arbiter
            sc_out<bool>  uvm_master_rreq;  // read request
            sc_out<bool>  uvm_master_wreq;  // write request
            sc_out<int>   uvm_master_addr;  // address
            sc_out<int>   uvm_master_wdata; // write data
            sc_in<int>    uvm_master_rdata; // read data
            sc_in<bool>   uvm_master_ack;   // acknowledge

            foreign_module_master(sc_module_name name, const char* hdl_name):
                uvm_master_clk("clk"),
                uvm_master_rst("rst"),
                uvm_master_req("req"),
                uvm_master_gnt("gnt"),
                uvm_master_rreq("rreq"),
                uvm_master_wreq("wreq"),
                uvm_master_addr("addr"),
                uvm_master_wdata("wdata"),
                uvm_master_rdata("rdata"),
                uvm_master_ack("ack")
            {
                std::string tmp = "";
                tmp += name;

                // the class sv_uvm_master is generated by Synopsys tool, see code below
                vcs_uvm_master = new sv_uvm_master(name, tmp.c_str());

                //Input ports
                vcs_uvm_master->clk(uvm_master_clk);
                vcs_uvm_master->rst(uvm_master_rst);
                vcs_uvm_master->gnt(uvm_master_gnt);
                vcs_uvm_master->rdata(uvm_master_rdata);
                vcs_uvm_master->ack(uvm_master_ack);

                //Output ports
                vcs_uvm_master->req(uvm_master_req);
                vcs_uvm_master->rreq(uvm_master_rreq);
                vcs_uvm_master->wreq(uvm_master_wreq);
                vcs_uvm_master->addr(uvm_master_addr);
                vcs_uvm_master->wdata(uvm_master_wdata);
            }

            ~foreign_module_master() {}

            sv_uvm_master *vcs_uvm_master;
        };
#     else
#         error "Cannot compile for VCS and NC_SYSTEMC simultaniously!"
#     endif
#   else
#     error "Cannot compile for VCS and MTI_SYSTEMC simultaniously!"
#   endif
#else
#   error "Must select one of the three simulation environments: VCS (vcs) MTI_SYSTEMC (Questa) or
NC_SYSTEMC (ncsim)!"
#endif
```

Wrapper generated by Synopsys tool:

```
// SystemC wrapper header for DKI connect HDL model import
// This file must be compiled with other SystemC files

#ifndef sv_uvm_master_h_
#define sv_uvm_master_h_

#include "systemc.h"
#include <string.h>
#include "cosim/bf/hdl_connect_v.h"
#include "cosim/bf/VcsDesign.h"
#include "cosim/bf/snps_hdl_param.h"

extern "C" unsigned int* Msv_uvm_master_1(unsigned int*, char*);
extern "C" std::string BF_get_hdl_name(SC_CORE sc_object *);


struct sv_uvm_master : sc_module {
   VcsModel    *vcsModel;
   VcsInstance *vcsInstance;

   //Input ports
   sc_in<bool>  clk;
   sc_in<bool>  rst;
   sc_in<bool>  gnt;
   sc_in<int>  rdata;
   sc_in<bool>  ack;

   //Output ports
   sc_out<bool>  req;
   sc_out<bool>  rreq;
   sc_out<bool>  wreq;
   sc_out<int>  addr;
   sc_out<int>  wdata;

   sc_signal<bool> *sig_clk;
   sc_signal<bool> *sig_rst;
   sc_signal<bool> *sig_gnt;
   sc_signal<int> *sig_rdata;
   sc_signal<bool> *sig_ack;

   sc_signal<bool> *sig_req;
   sc_signal<bool> *sig_rreq;
   sc_signal<bool> *sig_wreq;
   sc_signal<int> *sig_addr;
   sc_signal<int> *sig_wdata;

   // Parameters
   SC_SNPS hdl_param<std::string> str_param;

   SC_HAS_PROCESS(sv_uvm_master);
   sv_uvm_master(sc_module_name modelName): sc_module(modelName), HDL_PARAM(str_param, "MASTER")     {
      sig_clk = new sc_signal<bool>("sig_clk");
      sig_rst = new sc_signal<bool>("sig_rst");
      sig_gnt = new sc_signal<bool>("sig_gnt");
      sig_rdata = new sc_signal<int>("sig_rdata");
      sig_ack = new sc_signal<bool>("sig_ack");

      sig_req = new sc_signal<bool>("sig_req");
      sig_rreq = new sc_signal<bool>("sig_rreq");
      sig_wreq = new sc_signal<bool>("sig_wreq");
      sig_addr = new sc_signal<int>("sig_addr");
      sig_wdata = new sc_signal<int>("sig_wdata");

      HDL_MODULE("sv_uvm_master", name(), basename());
      SC_METHOD(sv_uvm_master_clk_action); sensitive << clk; snps_sysc_mark_last_create_process_as_internal();
      SC_METHOD(sv_uvm_master_rst_action); sensitive << rst; snps_sysc_mark_last_create_process_as_internal();
      SC_METHOD(sv_uvm_master_gnt_action); sensitive << gnt; snps_sysc_mark_last_create_process_as_internal();
      SC_METHOD(sv_uvm_master_rdata_action); sensitive << rdata; snps_sysc_mark_last_create_process_as_internal();
      SC_METHOD(sv_uvm_master_ack_action); sensitive << ack; snps_sysc_mark_last_create_process_as_internal();

      SC_METHOD(sv_uvm_master_req_action); sensitive << *sig_req; snps_sysc_mark_last_create_process_as_internal();
      SC_METHOD(sv_uvm_master_rreq_action); sensitive << *sig_rreq; snps_sysc_mark_last_create_process_as_internal();
      SC_METHOD(sv_uvm_master_wreq_action); sensitive << *sig_wreq; snps_sysc_mark_last_create_process_as_internal();
      SC_METHOD(sv_uvm_master_addr_action); sensitive << *sig_addr; snps_sysc_mark_last_create_process_as_internal();
      SC_METHOD(sv_uvm_master_wdata_action); sensitive << *sig_wdata; snps_sysc_mark_last_create_process_as_internal();

      std::string clk_string = name();
      clk_string += "_R_clk";
      alterString(clk_string, basename());
      hdl_connect_v(*sig_clk, clk_string.c_str(), HDL_OUTPUT, HDL_vcs);

      std::string rst_string = name();
      rst_string += "_R_rst";
      alterString(rst_string, basename());
      hdl_connect_v(*sig_rst, rst_string.c_str(), HDL_OUTPUT, HDL_vcs);

      std::string gnt_string = name();
      gnt_string += "_R_gnt";
      alterString(gnt_string, basename());
      hdl_connect_v(*sig_gnt, gnt_string.c_str(), HDL_OUTPUT, HDL_vcs);
```

```
    std::string rdata_string = name();
    rdata_string += "_R_rdata";
    alterString(rdata_string, basename());
    hdl_connect_v(*sig_rdata, rdata_string.c_str(), HDL_OUTPUT, HDL_vcs);

    std::string ack_string = name();
    ack_string += "_R_ack";
    alterString(ack_string, basename());
    hdl_connect_v(*sig_ack, ack_string.c_str(), HDL_OUTPUT, HDL_vcs);

    std::string req_string = name();
    alterString(req_string, basename());
    req_string += ".req";
    hdl_connect_v(*sig_req, req_string.c_str(), HDL_INPUT, HDL_vcs);

    std::string rreq_string = name();
    alterString(rreq_string, basename());
    rreq_string += ".rreq";
    hdl_connect_v(*sig_rreq, rreq_string.c_str(), HDL_INPUT, HDL_vcs);

    std::string wreq_string = name();
    alterString(wreq_string, basename());
    wreq_string += ".wreq";
    hdl_connect_v(*sig_wreq, wreq_string.c_str(), HDL_INPUT, HDL_vcs);

    std::string addr_string = name();
    alterString(addr_string, basename());
    addr_string += ".addr";
    hdl_connect_v(*sig_addr, addr_string.c_str(), HDL_INPUT, HDL_vcs);

    std::string wdata_string = name();
    alterString(wdata_string, basename());
    wdata_string += ".wdata";
    hdl_connect_v(*sig_wdata, wdata_string.c_str(), HDL_INPUT, HDL_vcs);

    vcsModel = VcsDesign::getDesignInstance()->addModel("sv_uvm_master");
    vcsInstance = vcsModel->addInstance(name());
    vcsInstance->setScObj(this);
    vcsInstance->setNames("sv_uvm_master", name(), basename());
    vcsInstance->addPort("clk",  VcsPort::INPUT_PORT, 1, 0);
    vcsInstance->addPort("rst",  VcsPort::INPUT_PORT, 1, 0);
    vcsInstance->addPort("gnt",  VcsPort::INPUT_PORT, 1, 0);
    vcsInstance->addPort("rdata",  VcsPort::INPUT_PORT, 32, 0);
    vcsInstance->addPort("ack",  VcsPort::INPUT_PORT, 1, 0);
    vcsInstance->addPort("req",  VcsPort::OUTPUT_PORT, 1, 0);
    vcsInstance->addPort("rreq",  VcsPort::OUTPUT_PORT, 1, 0);
    vcsInstance->addPort("wreq",  VcsPort::OUTPUT_PORT, 1, 0);
    vcsInstance->addPort("addr",  VcsPort::OUTPUT_PORT, 32, 0);
    vcsInstance->addPort("wdata",  VcsPort::OUTPUT_PORT, 32, 0);
    vcsInstance->addParameter("str_param", VcsParam::HDL_STRING, "\"MASTER\"");
};

sv_uvm_master(sc_module_name modelName, const std::string &str_param_): sc_module(modelName)

    , HDL_PARAM(str_param, "MASTER")   {
    sig_clk = new sc_signal<bool>("sig_clk");
    sig_rst = new sc_signal<bool>("sig_rst");
    sig_gnt = new sc_signal<bool>("sig_gnt");
    sig_rdata = new sc_signal<int>("sig_rdata");
    sig_ack = new sc_signal<bool>("sig_ack");

    sig_req = new sc_signal<bool>("sig_req");
    sig_rreq = new sc_signal<bool>("sig_rreq");
    sig_wreq = new sc_signal<bool>("sig_wreq");
    sig_addr = new sc_signal<int>("sig_addr");
    sig_wdata = new sc_signal<int>("sig_wdata");

    HDL_MODULE("sv_uvm_master", name(), basename());
    SC_METHOD(sv_uvm_master_clk_action); sensitive << clk; snps_sysc_mark_last_create_process_as_internal();
    SC_METHOD(sv_uvm_master_rst_action); sensitive << rst; snps_sysc_mark_last_create_process_as_internal();
    SC_METHOD(sv_uvm_master_gnt_action); sensitive << gnt; snps_sysc_mark_last_create_process_as_internal();
    SC_METHOD(sv_uvm_master_rdata_action); sensitive << rdata; snps_sysc_mark_last_create_process_as_internal();
    SC_METHOD(sv_uvm_master_ack_action); sensitive << ack; snps_sysc_mark_last_create_process_as_internal();

    SC_METHOD(sv_uvm_master_req_action); sensitive << *sig_req; snps_sysc_mark_last_create_process_as_internal();
    SC_METHOD(sv_uvm_master_rreq_action); sensitive << *sig_rreq; snps_sysc_mark_last_create_process_as_internal();
    SC_METHOD(sv_uvm_master_wreq_action); sensitive << *sig_wreq; snps_sysc_mark_last_create_process_as_internal();
    SC_METHOD(sv_uvm_master_addr_action); sensitive << *sig_addr; snps_sysc_mark_last_create_process_as_internal();
    SC_METHOD(sv_uvm_master_wdata_action); sensitive << *sig_wdata; snps_sysc_mark_last_create_process_as_internal();

    std::string clk_string = name();
    clk_string += "_R_clk";
    alterString(clk_string, basename());
    hdl_connect_v(*sig_clk, clk_string.c_str(), HDL_OUTPUT, HDL_vcs);

    std::string rst_string = name();
    rst_string += "_R_rst";
    alterString(rst_string, basename());
    hdl_connect_v(*sig_rst, rst_string.c_str(), HDL_OUTPUT, HDL_vcs);
```

```
        std::string gnt_string = name();
        gnt_string += "_R_gnt";
        alterString(gnt_string, basename());
        hdl_connect_v(*sig_gnt, gnt_string.c_str(), HDL_OUTPUT, HDL_vcs);
        std::string rdata_string = name();
        rdata_string += "_R_rdata";
        alterString(rdata_string, basename());
        hdl_connect_v(*sig_rdata, rdata_string.c_str(), HDL_OUTPUT, HDL_vcs);
        std::string ack_string = name();
        ack_string += "_R_ack";
        alterString(ack_string, basename());
        hdl_connect_v(*sig_ack, ack_string.c_str(), HDL_OUTPUT, HDL_vcs);
        std::string req_string = name();
        alterString(req_string, basename());
        req_string += ".req";
        hdl_connect_v(*sig_req, req_string.c_str(), HDL_INPUT, HDL_vcs);
        std::string rreq_string = name();
        alterString(rreq_string, basename());
        rreq_string += ".rreq";
        hdl_connect_v(*sig_rreq, rreq_string.c_str(), HDL_INPUT, HDL_vcs);
        std::string wreq_string = name();
        alterString(wreq_string, basename());
        wreq_string += ".wreq";
        hdl_connect_v(*sig_wreq, wreq_string.c_str(), HDL_INPUT, HDL_vcs);
        std::string addr_string = name();
        alterString(addr_string, basename());
        addr_string += ".addr";
        hdl_connect_v(*sig_addr, addr_string.c_str(), HDL_INPUT, HDL_vcs);
        std::string wdata_string = name();
        alterString(wdata_string, basename());
        wdata_string += ".wdata";
        hdl_connect_v(*sig_wdata, wdata_string.c_str(), HDL_INPUT, HDL_vcs);

        vcsModel = VcsDesign::getDesignInstance()->addModel("sv_uvm_master");
        vcsInstance = vcsModel->addInstance(name());
        vcsInstance->setScObj(this);
        vcsInstance->setNames("sv_uvm_master", name(), basename());
        vcsInstance->addPort("clk",  VcsPort::INPUT_PORT, 1, 0);
        vcsInstance->addPort("rst",  VcsPort::INPUT_PORT, 1, 0);
        vcsInstance->addPort("gnt",  VcsPort::INPUT_PORT, 1, 0);
        vcsInstance->addPort("rdata",  VcsPort::INPUT_PORT, 32, 0);
        vcsInstance->addPort("ack",  VcsPort::INPUT_PORT, 1, 0);
        vcsInstance->addPort("req",  VcsPort::OUTPUT_PORT, 1, 0);
        vcsInstance->addPort("rreq",  VcsPort::OUTPUT_PORT, 1, 0);
        vcsInstance->addPort("wreq",  VcsPort::OUTPUT_PORT, 1, 0);
        vcsInstance->addPort("addr",  VcsPort::OUTPUT_PORT, 32, 0);
        vcsInstance->addPort("wdata",  VcsPort::OUTPUT_PORT, 32, 0);
        vcsInstance->addParameter("str_param", VcsParam::HDL_STRING, "\"MASTER\"");
        str_param(str_param_);
    };

    void sv_uvm_master_clk_action()
    { sig_clk->write(clk.read());
    }
    void sv_uvm_master_rst_action()
    { sig_rst->write(rst.read());
    }
    void sv_uvm_master_gnt_action()
    { sig_gnt->write(gnt.read());
    }
    void sv_uvm_master_rdata_action()
    { sig_rdata->write(rdata.read());
    }
    void sv_uvm_master_ack_action()
    { sig_ack->write(ack.read());
    }

    void sv_uvm_master_req_action()
    { req.write((*sig_req).read());
    }
    void sv_uvm_master_rreq_action()
    { rreq.write((*sig_rreq).read());
    }
    void sv_uvm_master_wreq_action()
    { wreq.write((*sig_wreq).read());
    }
    void sv_uvm_master_addr_action()
    { addr.write((*sig_addr).read());
    }
    void sv_uvm_master_wdata_action()
    { wdata.write((*sig_wdata).read());
    }

    const char *kind() const { return "dki_module_verilog"; }
};


#endif
```

9. Appendix B

Compile and run script used to execute the SystemC-UVMC-SystemVerilog mixed language simulation.

```
vlogan -cpp ${CPP} -ntb_opts uvm-1.1 -sverilog
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ubus_if.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ubus_transfer_pkg.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ifs_command_monitor_pkg.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ubus_master_driver_pkg.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ubus_master_monitor_pkg.VCS.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ubus_master_sequencer_pkg.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ubus_master_seq_pkg.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 \
    +incdir+${UVMC_HOME}/src/connect/sv ${UVMC_HOME}/src/connect/sv/uvmc_pkg.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ifs_command_master_agent_pkg.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ubus_example_master_seq_pkg.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/ifs_command_master_agent_pkg.sv
vlogan -cpp ${CPP} -sverilog +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/uvm_master_module.sv
vlogan -cpp ${CPP} -cc ${CC} -sysc -sverilog -sc_model sv_uvm_master \
    -sc_portmap masterVerilogToSystemCport.map \
    +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/uvm_master_module.sv
vlogan -cpp ${CPP} -cc ${CC} -sysc -sverilog -sc_model sv_uvm_test \
    +incdir+${VCS_HOME}/etc/uvm-1.1 ../tb_uvm/uvm_test_module.sv

vcs -cpp ${CPP} -cc ${CC} -cflags -g -sysc -sverilog -sysc=scv -sysc=deltasync \
    -ntb_opts uvm-1.1 -debug_all -timescale=1ps/1ps \
    -cflags -m32 -cflags -DVCS -cflags -DUVMC_NO_COUT_STL_LIST -cflags -DSC_INCLUDE_DYNAMIC_PROCESSES \
    -cflags -DINCLUDE_UVM_AGENT -cflags -I. -cflags -I${VCS_HOME}/include/scv-2.0 \
    -cflags -I${VCS_HOME}/etc/systemc/tlm/include/tlm    \
    -cflags -I${VCS_HOME}/etc/systemc/tlm/include/tlm/tlm_utils \
    -cflags -I${UVMC_HOME}/src/connect/sc -cflags -I${IFS_HOME}/Include \
    -cflags -I${IFS_HOME}/Source/spirit-1.6.1 -cflags -I${IFS_HOME}/Source/spirit-1.6.1/miniboost \
    -cflags -I../tb_systemc/ -cflags -I../rtl_systemc/ \
       ${UVMC_HOME}/src/connect/sc/uvmc.cpp \
       ${IFS_HOME}/Source/*.cpp \
       ../rtl_systemc/arbiter.cpp \
       ../tb_systemc/clk_ifs.cpp \
       ../tb_systemc/ictrl.cpp \
       ../tb_systemc/master.cpp \
       ../tb_systemc/slave.cpp \
       ../tb_systemc/wrap_uvm_master.cpp \
       ../tb_systemc/main.cpp

./simv
```

10.Appendix C

Wrapper UVM connector is a UVM component derived from the UVM monitor class. It receives the TLM message and creates an according sequence item. The sequence item is forwarded through the UVM sequencer to the UVM driver.

```
package ifs_command_monitor_pkg;

    import uvm_pkg::*;
    `include "uvm_macros.svh"

    import ubus_transfer_pkg::ubus_transfer;
    import ubus_transfer_pkg::wrap_uvm_command;
    import ubus_transfer_pkg::NOP;
    import ubus_transfer_pkg::READ;
    import ubus_transfer_pkg::WRITE;

    class ifs_command_monitor extends uvm_monitor;
        // uvm_tlm_b_target_socket #(ifs_command_monitor) in;
        uvm_tlm_b_target_socket #(ifs_command_monitor, wrap_uvm_command) in;
        uvm_blocking_peek_imp#(ubus_transfer,ifs_command_monitor) ifs_command_monitor_addr_ph_imp;

        // The following property holds the transaction information currently
        // created from detected changes in the DB.
        protected ubus_transfer command_from_TLM_sc;

        // monitor notifier that the address phase (and full item) has been collected
        protected event address_phase_grabbed;
        protected event DEBUGevent;

        // Provide implementations of virtual methods such as get_type_name and create
        `uvm_component_utils_begin(ifs_command_monitor)
        `uvm_component_utils_end

        // new - constructor
        function new(string name, uvm_component parent=null);
            super.new(name, parent);
            in = new("in", this);
            command_from_TLM_sc = new();
            ifs_command_monitor_addr_ph_imp = new("ifs_command_monitor_addr_ph_imp", this);
        endfunction : new

        // task called via 'in' socket
```

```
       virtual task b_transport (wrap_uvm_command t, uvm_tlm_time delay);
          void'(this.begin_tr(command_from_TLM_sc));

          `uvm_info(get_type_name(),
             $sformatf("********* ifs_command_monitor::b_transport(): SC-TLM communication received: cmd -
%h, parameters - %p",
             t.cmd, t.parameters), UVM_MEDIUM);

          // Here we decode the TLM to generate the correct sequence_item
          // from it.
          case (t.cmd)
             WRITE: write_transaction(t.parameters);
             READ:  read_transaction(t.parameters);
          endcase

          this.end_tr(command_from_TLM_sc);
          // Inform task peek about his
          -> address_phase_grabbed;
       endtask

       function void write_transaction(string parameters[]);
          if (parameters.size !== 2) begin
             `uvm_info(get_type_name(),  "*********  ifs_command_monitor::write_transaction():  wrong  number
of parameters for write transaction!",UVM_MEDIUM);
          end
          else begin
             command_from_TLM_sc.read_write = WRITE;
             command_from_TLM_sc.addr = parameters[0].atoi();
             command_from_TLM_sc.data = parameters[1].atoi();
          end
       endfunction : write_transaction

       function void read_transaction(string parameters[]);
          if (parameters.size !== 2) begin
             `uvm_info(get_type_name(), "********* ifs_command_monitor::read_transaction(): wrong number of
parameters for read transaction!",UVM_MEDIUM);
          end
          else begin
             command_from_TLM_sc.read_write = READ;
             command_from_TLM_sc.addr = parameters[0].atoi();
             command_from_TLM_sc.data = parameters[1].atoi();
          end
       endfunction : read_transaction

       task peek(output ubus_transfer trans);
          @address_phase_grabbed;
          `uvm_info(get_type_name(), $sformatf("********* ifs_command_monitor::peek(): Informing driver to
drive cmd-%d, addr-%d, data-%d!",
             command_from_TLM_sc.read_write,
             command_from_TLM_sc.addr,
             command_from_TLM_sc.data),UVM_MEDIUM);
          trans = command_from_TLM_sc;
       endtask : peek
    endclass : ifs_command_monitor
endpackage : ifs_command_monitor_pkg
```