

# Connecting a Company's 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

**ABSTRACT**

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.

Based on VHDL in the past, our IFS verification methodology was also implemented in SystemC (SC) [2] and covers Analog Mixed-Signal (AMS) [1] [4], and Matlab/Simulink [3] today. In this work we proceed with concepts of UVM [9] and show how UVM components are instantiable in our SC test environment to verify designs specified in VHDL (-AMS), SystemC (-AMS), Verilog (-AMS) or any language a mixed-language simulation environment exists for. Our work does not depend on proprietary technology, but is applicable to any SC based environment.

## 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 [1], 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 [1] to SC [2], Matlab/Simulink [3] 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.

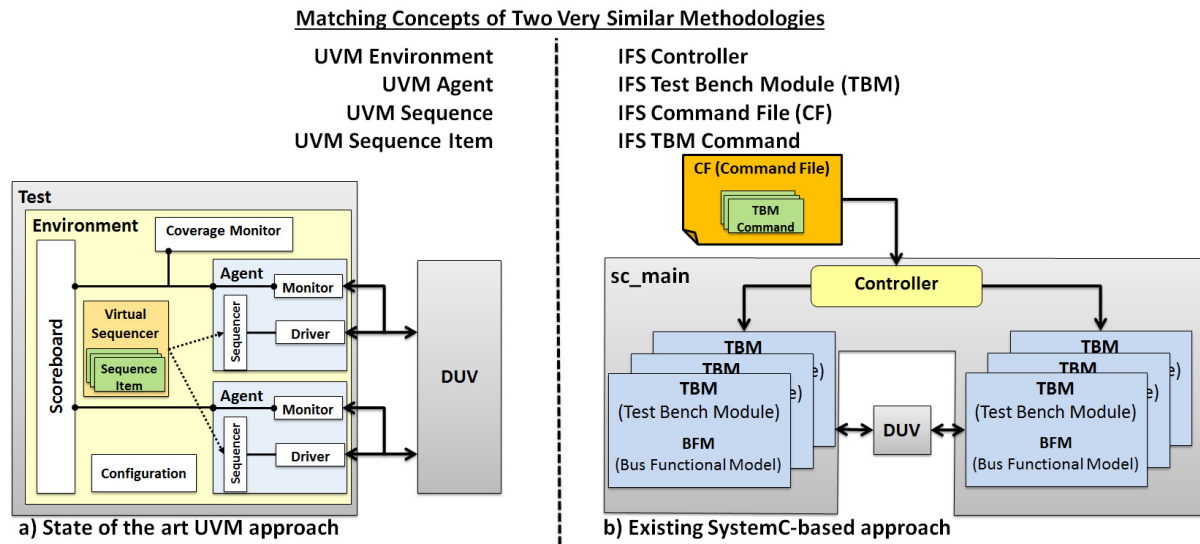


Figure 1: Juxtaposition of a) UVM and b) IFS.

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 Mentor Graphics Questa and Cadence Incisive (Chapter 5). The document concludes in Chapter 6.

## 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.

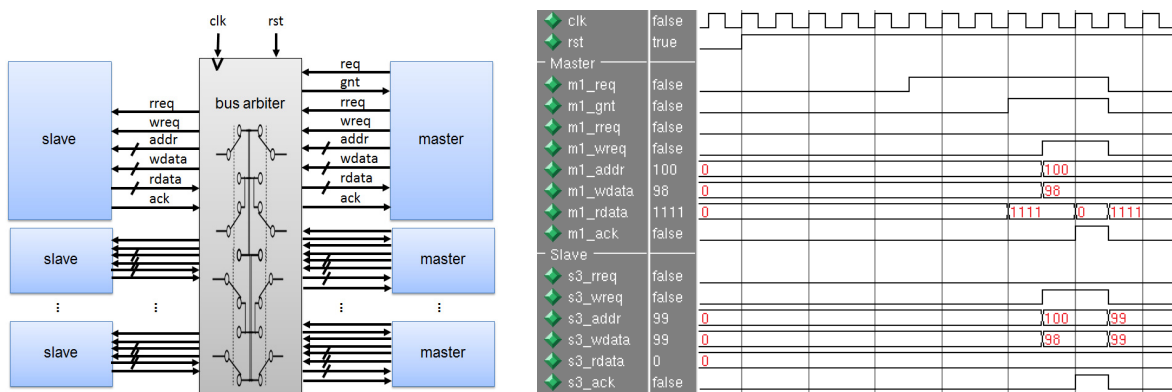


Figure 2: Switched bus arbiter example as DUV for the experiment of this work.

### 3. Integrated Functional Verification Script Environment and Bus-Arbitrator Example

A first introduction into Integrated Functional Verification Script Environment (IFS) is to be found in [1] and [2]. 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.

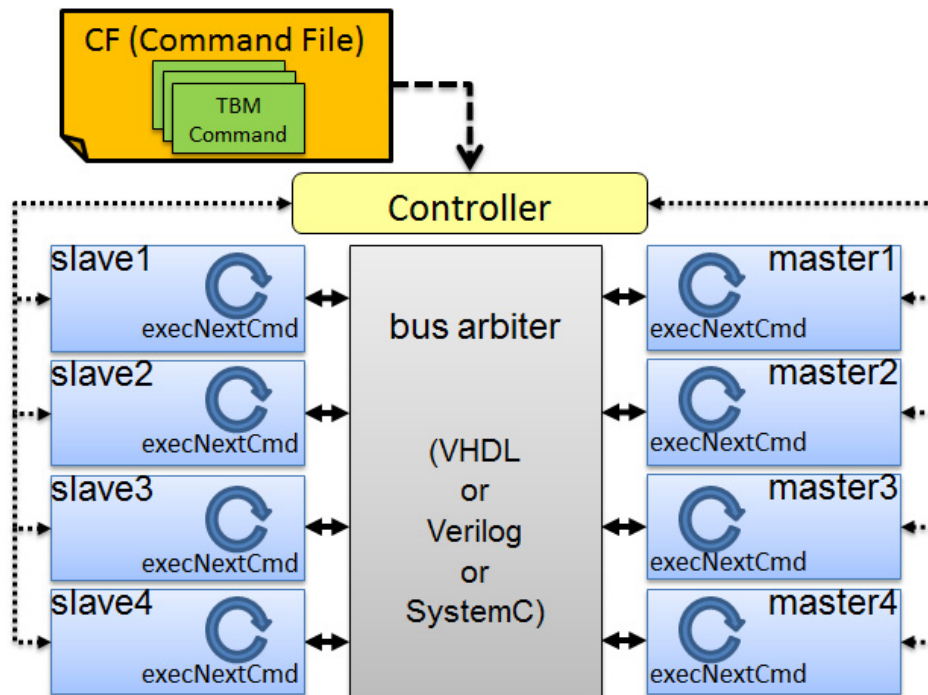


Figure 3: IFS simulation environment for a bus arbiter test bench.

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 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 reimplementaion 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 for SC test benches.

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.

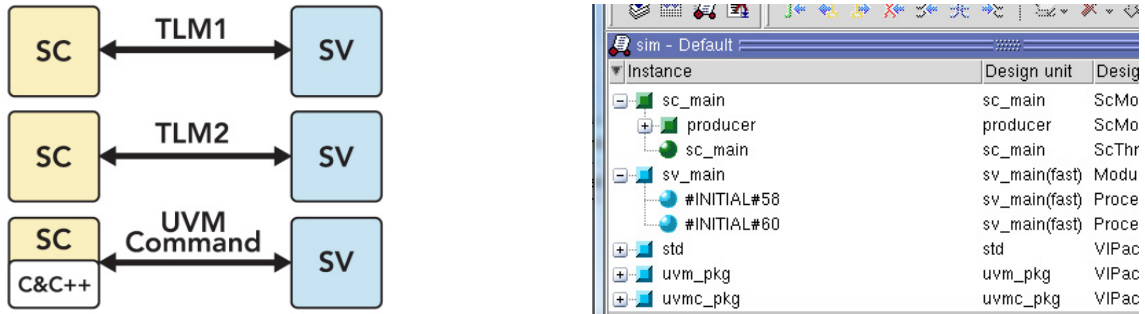


Figure 4: UVMC as link for TLM and commands between SC and SV (source: UVMC documentation).

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] and [11]) 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.

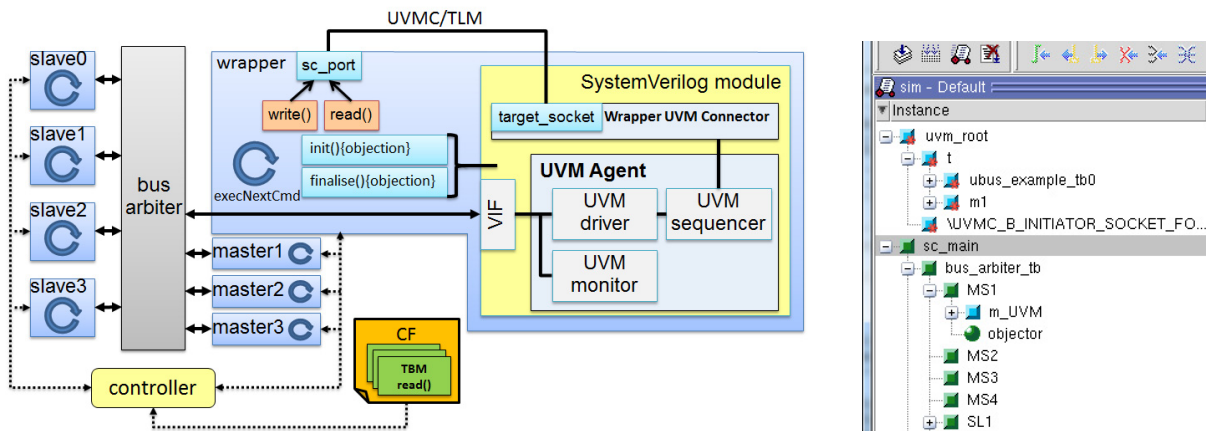


Figure 5: SC instantiating UVM in wrapper module using the foreign module interface.

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 Mentor Graphics' Questa simulator (`sc_foreign_module` [10]) and Cadence's Incisive (`ncsc_foreign_module` [11]) with success. 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 two tool setups. The first setup consist of Mentor Graphics Questa 10.1c, UVM 1.1b and UVMC v2.2. The second setup utilized Cadence Incisive 13.10-s005 We believe that any setup complying with IEEE 1666 SC and IEEE 1800 SV should work. For example, the author of [5] does not report issues in using UVMC with Synopsys VCS.

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 11 document the elaboration phase.

```

1:  irun(64): 13.10-s005: (c) Copyright 1995-2013 Cadence Design Systems, Inc.
2:  Loading snapshot worklib.bus_arbiter_tb:sc_module ..... Done
3:  ***** foreign_module_master::CTOR(): Elaborating foreign module: sv_uvm_master
4:  ***** wrap_uvm_master::CTOR(): Connecting TLM port
5:  Connecting an SC-side proxy chan for 'bus_arbiter_tb.MS1.port_10' with lookup string
   'sc_wrap_MS1' for later connection with SV
6:  INFO (0 s) [bus_arbiter_tb.MS1] Registered module 'bus_arbiter_tb.MS1'
7:  ...(also MS2 to MS4)
8:  INFO (0 s) [bus_arbiter_tb.SL1] Registered module 'bus_arbiter_tb.SL1'
9:  ...(also SL2 to SL4)
10: INFO (0 s) [bus_arbiter_tb.CLK] Registered module 'bus_arbiter_tb.CLK'
11: INFO (0 s) Loading script: 'control.cmd'

```

Run of simulation starts in line 12, where UVM begins to traverse through its simulation phases build, connect and finally entering the main run phase with line 23. 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 5 above, UVM line 22 below).

```

12: ncsim> run
13: -----
14: CDNS-UVM-1.1d (13.10-s005)
15: (C) 2007-2013 Mentor, Cadence, Synopsys, Cypress Semiconductor
16: -----
17: UVM_INFO ../tb_uvm/uvm_master_module.sv(57) @ 0: reporter [sv_uvm_master] *****
   sv_uvm_master: Initialising instance sc_wrap_MS1 and its virtual interface.
18: -----
19: UVMC-2.2
20: (C) 2009-2012 Mentor Graphics Corporation
21: -----
22: Registering SV-side 'sc_wrap_MS1.ifs_monitor.in' and lookup string 'sc_wrap_MS1' for later
   connection with SC
23: UVM_INFO @ 0 ns: reporter [RNTST] Running test ...

```

The run phase starts and the wrapper raises an objection for the UVM run phase (line 24). UVMC connects the SC and UVM TLM ports by referring to their label (line 25). In line 31 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 32, the wrapper UVM

connector receives the TLM message and creates a sequence item 33. The UVM agent's driver consumes the sequence item and applies according signal events to the DUV (line 34). The SC slave TBM responds to the signal events (line 35).

```

24: ***** wrap_uvm_master::initialiseModule(): Raising objection for UVM phase 'run'!
25: Connected SC-side 'bus_arbiter_tb.MS1.port_10' to SV-side 'sc_wrap_MS1.ifs_monitor.in'
26: 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...
27: [bus_arbiter_tb.CLK,100 ns] reset gets passiv
28: [bus_arbiter_tb.SL1,100 ns] set slave_offset = 300
29: ... (configuring SL1 to SL4)
30: 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!
31: ***** wrap_uvm_master::Write(): Sending payload '{cmd:2 parameters:100 98}' to MS1.socket
    at time 150
32: 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"}'
33: 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!
34: 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!
35: [bus_arbiter_tb.SL3,190 ns] Write (Address: 100, Value: 98)
36: 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 37 to 47).

```

37: UVM_INFO      ../tb_uvm/ubus_master_monitor_pkg.sv(137)      @      195      ns:      sc_wrap_MS1.monitor
    [sc_wrap_MS1.monitor] Master transfer collected :
38: -----
39: ubus_transfer_inst  ubus_transfer      -      @6690
40: read_write          ubus_read_write_enum 32      WRITE
41: addr                integral            32      'h64
42: data                integral            32      'h62
43: master              string              11      sc_wrap_MS1
44: slave                string              0       ""
45: begin_time           time                64      180 ns
46: end_time             time                64      195 ns
47: -----

```

Lines 48 to 51 demonstrate that the other TBM masters operate in parallel just as well issuing write commands that are answered by the TBM slaves.

```

48: [bus_arbiter_tb.MS2,820 ns] Write (Address: 16, Value: 32)
49: [bus_arbiter_tb.SL4,830 ns] Write (Address: 16, Value: 32)
50: [bus_arbiter_tb.MS4,1010 ns] Write (Address: 100, Value: 98)
51: [bus_arbiter_tb.SL3,1020 ns] Write (Address: 100, Value: 98)

```

On reaching the quit command the run-phase objection is being dropped (line 54) and the simulation terminates.

```

52: INFO (2030 ns) SIMULATION END FROM COMMAND FILE
53: INFO (2030 ns) Exiting simulation.
54: ***** wrap_uvm_master::finaliseModule(): Dropping objection for UVM phase 'run'!
55: SC simulation stopped by user.
56: SystemC : SystemC stopped at time 2030
57: ncsim> exit

```

## 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 two simulation environments we used. As literature [5] suggests that UVMC works with other simulation environments, too, it is expectable that our approach will work here as well.

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.

- [1] 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 (Kurztitle: VeronA)”, Schlussbericht zur BMBF-Förderinitiative IKT2020, 2009.
- [2] R. Lissel and J. Gerlach, “Introducing new verification methods into a company's design flow: an industrial user's point of view”, DATE 2007.
- [3] K. Hylla, J.-H. Oetjens and W. Nebel, “Using SystemC for an Extended MATLAB/Simulink Verification Flow”, FDL 2008.
- [4] 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.
- [5] T. Leitner, A. Harapanahalli and O. Bell, “Boosting VP and RTL verification by leveraging a reusable UVM environment”, SNUG Germany 2014.
- [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-Gritschneider, 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] Mentor, “Questa® SIM User's Manual Including Support for Questa SV/AFV”, 2012.
- [11] Cadence, “SystemC Simulation User Guide”, 2012.