2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

UVM-Multi-Language Hands-On

Integrating a System-C model into a SV-UVM testbench

Thorsten Dworzak, Verilab Gmbh, Munich, Germany (thorsten.dworzak@verilab.com)

Angel Hidalga, Infineon Technologies AG, Munich, Germany (angel.hidalga@infineon.com)

***Abstract*—The Universal-Verification-Methodology Multi-language (UVM-ML) library is a simulator independent extension of the UVM and promises easy integration of different high-level verification languages. We applied it together with previously existing techniques to attach a SystemC model to different UVM SystemVerilog testbenches. The interface between the two worlds has been implemented using TLM2, DPI-C, and FMI. Over the course of this project we faced some obstacles and stumbling blocks concerning different aspects. By sharing our experience and some tips and hints, we hope to provide others with a smoother experience.**

Keywords: UVM-ML, SystemC, SystemVerilog, TLM2, DPI-C

I. INTRODUCTION

For the verification of an ARM CPU IP we developed a SystemC [2] model. It has been a joint development effort with the software team at the company. The model serves three different verification focuses.

First, it is used as a reference model for the DUT in a fully-featured UVM-SV testbench. The testbench provides stimulus generation, scoreboarding, and coverage. Stimulus generation and scoreboarding have their own C-model instance with a slightly different feature set. Both model instances are passive in that they receive CPU instructions from the testbench.

Second, it is used as a stand-alone instruction stream simulator (ISS), embedded in another UVM-SV testbench that mainly provides shared memory and allows running a set of self-checking assembler tests. This testbench can use either the model or the DUT as a drop-in component. The C-model in this case operates in active mode, i.e. it fetches CPU instructions from the shared memory.

Third, we expect that the C-model is going to be an integral part of a software simulator for firmware development. In this context the model needs to interact with other SystemC models (e.g. a memory system model).

II. REQUIREMENTS

A. *Requirements of the UVM testbench*

- Support constrained-random stimuli through UVM-SV-based instruction stream generator (ISG) as well as directed (self-checking) assembler/C tests from legacy environments.
- The ISG requires an exclusive instance of a non-time-consuming reference model to steer instruction sequence generation. We believe that every useful ISG implementation mandates some sort of reference model to avoid excessive creation of illegal CPU states and exceptions.

B. *Requirements of the C-model*

- ARM architecture and DUT specification compliance
- Automatic feature extraction from specifications (code generation)
- Support Linux/Windows cross-platform development (Microsoft Visual Studio/clang/gcc); the software team relies on the OSCI SystemC 2.3.1 library
- Use C++11 (ISO/IEC 14882:2011) wrapped in a few SystemC modules
- High performance
- Support all three use-cases with minimal code-overlap (C++ interfaces, base-classes and derived classes)
- Code partitioning should match the physical DUT hierarchy

- Support for more than one instance running at the same time calls for careful usage of static classes
- Matching model and DUT output (modelling of bus transfers and order thereof, CPU state, configuration registers)
- Must run in sync with the DUT, keeping the execution order; this requires a synchronization interface to match the CPU states (reset, instruction execution, exception handling, stacking, etc.)
- Instruction fetch interface either passive or active, supports TLM2 (Transaction-Level Modelling standard) [3] target and initiator socket
- Must support error injection to simulate faults (bus faults, security faults etc.)
- Reference model for the ISG must run ahead of the DUT in order to generate sensible sequences. This requires a state roll-back mechanism to support asynchronous exceptions, i.e. exceptions that are not immediately caused by the program execution flow

III. SOLUTION

We used the Cadence Incisive Simulator (IES) [1] with the UVM-ML package version 1.2. In this chapter we provide details on some selected implementation aspects that might be beneficial for the reader, regardless of the selected vendor.

A. Testbench Environment

Figure 1 illustrates the UVM environment containing two C-models. This is a standard UVM setup but it has

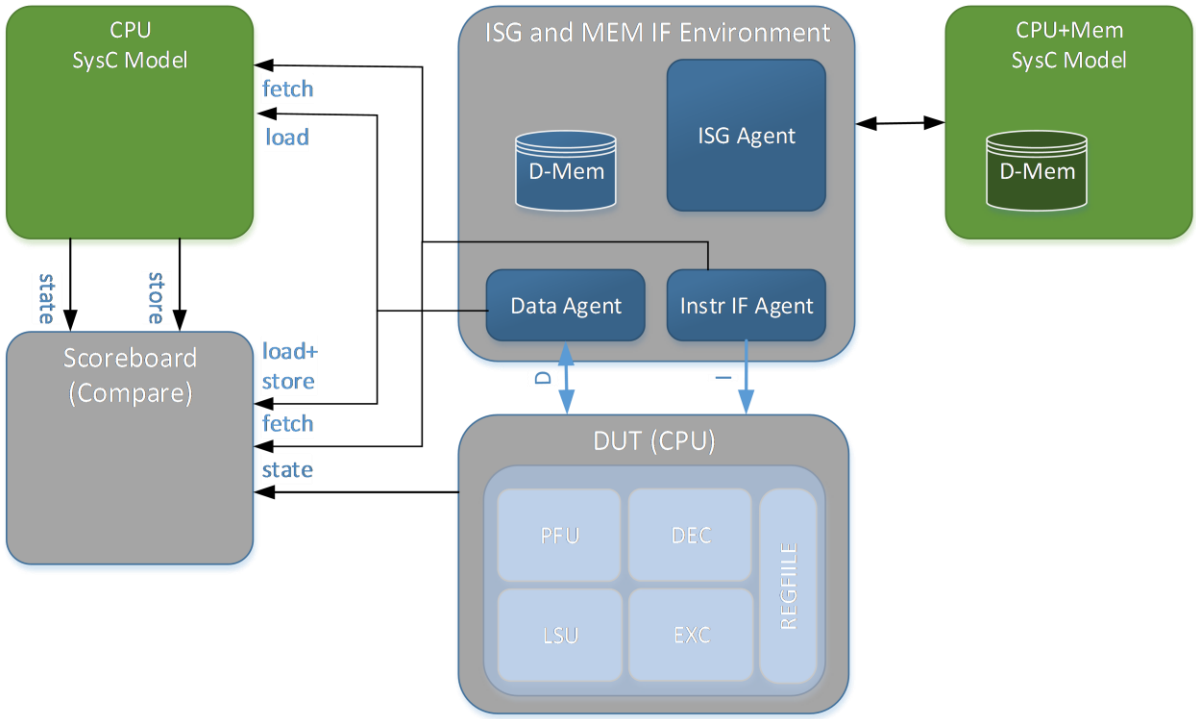

Figure 1: Testbench environment

some distinctive features that we want to highlight.

The ARM CPU is a von-Neumann-Architecture but has separate busses for data and instructions. On system-level, which is not the verification focus of this testbench, these connect to the same memory. To keep the architecture simple, we did not create a shared memory model - but the draw-back is that we can't simulate self-modifying code.

The sub-environment containing the ISG instantiates the C-model used to control the instruction sequence generation. It contains its own data memory, which can be accessed by the ISG through DPI calls. The ISG

generates an instruction and passes it for processing to this model. This environment also comprises the memory for load/store data of the DUT. For directed tests it is also necessary to provide instruction memory, which is embedded in a dedicated sequence that also reads the code images.

The scoreboard and the reference model are passive and receive instructions (fetch) and data (load + store) from the monitoring of the CPU busses and white-box probes monitoring internal register accesses. The monitoring agents also provide the scoreboard with CPU status information (state). The C-model calculates expected state information and store transactions which get compared in the scoreboard.

B. Stimuli Generation

The ISG is based on an instruction class inheriting from `uvm_sequence_item` that contains all properties of an instruction, e.g. name, opcode size, legal source/destination registers. For each instruction there is a derived class that refines these properties using constraints or direct assignments. These derived classes are generated from the specification. Figure 2 shows the UML diagram for the class describing a specific flavor of an ADC instruction.

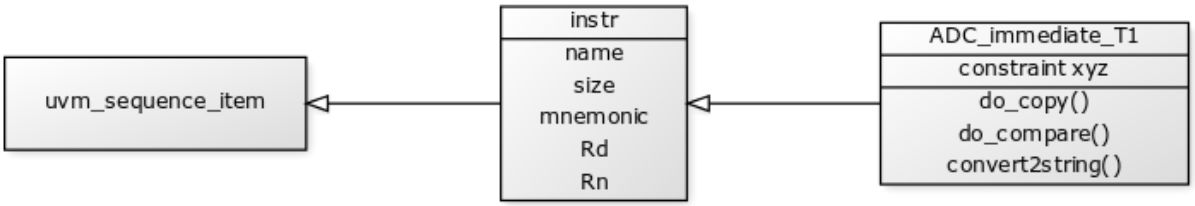

Figure 2: Instruction class diagram

An ISG sequence contains an instruction object and several fields to control the item generation, e.g. a pointer to the component that allows to get state information from the C-model.

C. OSCI vs. ncsc

OSCI is the reference implementation for the SystemC standard. It was used by the software team which does all of its development with Microsoft Visual Studio (MSVS). The verification team on the Linux side used the Cadence implementation which simplified the tool flow and offered sufficient compatibility to the OSCI implementation.

D. Linux vs. Windows

As the SV testbench was not available on the Windows side, the software team made extensive use of the freely available Google Test framework to develop unit tests. Furthermore, a number of Jenkins [4] projects were created to run code linters, check the build process and even run a gcc build to ensure code compatibility for both worlds. IES ships with its own gcc installation, so compatibility to this specific version was important. In general, gcc proved to be stricter regarding certain coding styles or C++ constructs than MSVS.

The software team was using a development flow based on Agile [5] principles. Working together with the verification team who followed a traditional waterfall model was sometimes challenging because of conflicting requirements. Being exposed to the tools and methodologies of the other team was a valuable learning experience though.

E. Language Interface Selection

The choice of TLM2 over TLM1 (Transaction-Level Modelling standard 1 and 2) was made due to ease-of-use and better support in SystemC. For TLM2 a generic payload for bus-like, memory-mapped protocols and support for automatic packing/un-packing and serialization/de-serialization is available. The data handling of the TLM1/TLM2 standard implemented in the UVM-ML package is not efficient in terms of performance though. The transaction data is copied into a structure, converted into a character stream, passed as a reference, and reconstructed - it implies at least 2 copies. Therefore we decided to use the DPI standard for asynchronous signals and complex data structures, e.g. CPU-state information. As indicated by its name, the “Direct Programming Interface” [6] allows referencing data structures from System Verilog to System-C and vice versa. The

disadvantage of DPI is that functions are declared global/static, which has to be considered. Figure 3 depicts the interfaces of one of the reference model versions. The following types of interfaces are applied:

- **TLM2 ports**: Busses, debug accesses and synchronization (cache interfaces, AHB peripheral bus, Special-function-register access etc.). The ports implement only blocking functions with and without time consumption, depending on the use-case.
- **DPI-C**: Reporting for CPU-Status, exceptions, interrupts, and asynchronous signals, like reset, alarms, etc.; mostly non-time-consuming.
- **FMI (Foreign Model Interface)**: The DUT re-used the C-model libraries to dump trace data (disassembly and state information). This was done from a legacy VHDL trace/debug component, thus we decided to use FMI. It has less overhead in terms of parameter-passing than VPI/VHPI (Verilog/VHDL Programming Interface), especially if there is no need to access simulator nodes.

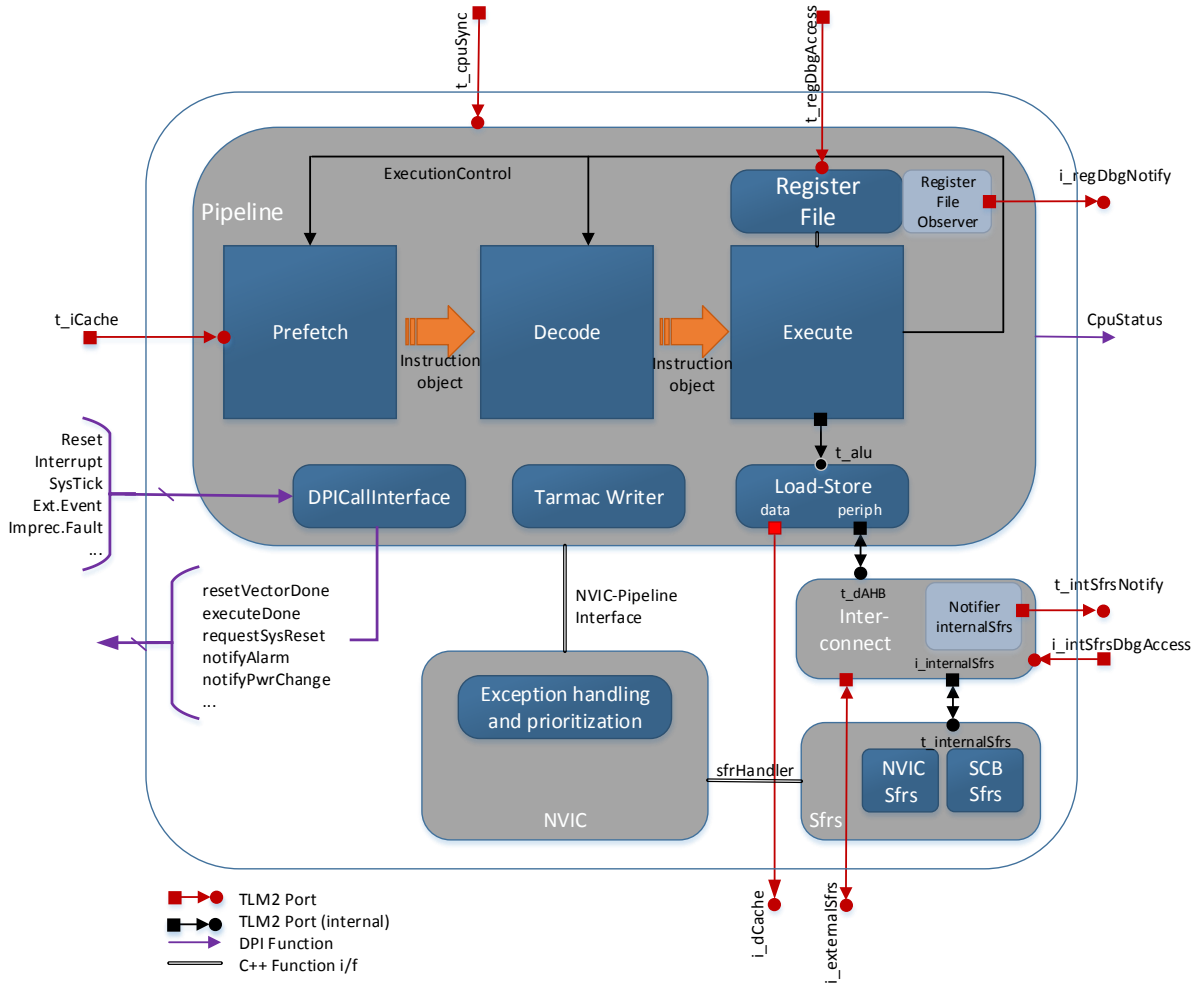

Figure 3: C-model interfaces

The TLM2 interfaces used on the SV-side are of type `uvm_tlm_b_<target|initiator>_socket`. Note that multi-sockets and passthrough sockets are currently not supported by the UVM-ML. For the data type we had to keep the default `uvm_tlm_generic_payload` which comes with a number of features as part of the UVM-ML

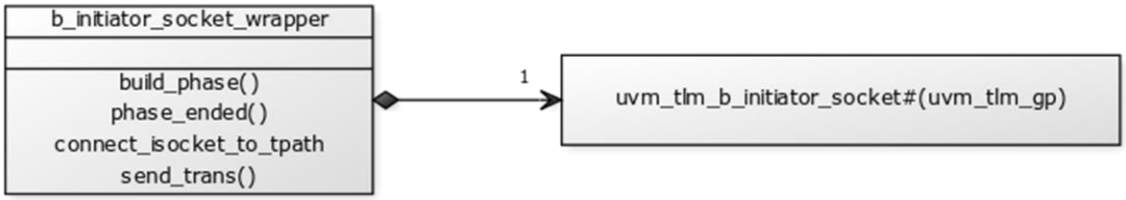

Figure 4: Wrapper class for blocking initiator socket

(see F). We created wrapper classes for the sockets in order to

- encapsulate basic functionality like registration and connection
- prevent name-clashing for the `b_transport()` function of the target socket (the function is implemented in the enclosing object). Figure 4 and Figure 5 show the UML relationship of the wrapper classes.

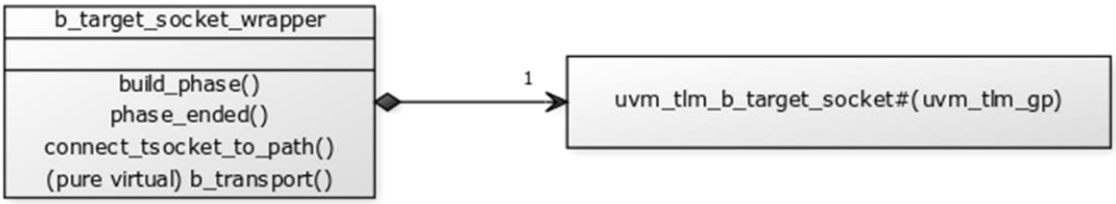

Figure 5: Wrapper class for blocking target socket

F. TLM2 Generic Payload and Extensions

The generic payload is the default data class of TLM2 sockets. It is suitable for memory-mapped bus-protocols. The automatic packing/serialization of the UVM-ML is only available for the base-type of the class. Any additional functionality that we needed has been implemented as static methods in a utility package, e.g. to make the transition from a 64-bit to a 32-bit bus, pack the payload data to an AMBA bus data vector or provide a customized printing method. One an example shown here is a clone function which extends the built-in clone function (see Figure 6) to copy the otherwise missing transaction id field.

```
function uvm_tlm_gp utils::clone(const ref uvm_tlm_gp rhs);
   $cast(clone, rhs.clone());
   assert(clone != null);
   clone.set_transaction_id(rhs.get_transaction_id());
endfunction
```

Figure 6: Copy function for tlm2 generic payload

We use the payload extension mechanism to transport additional information across the language boundary. A TLM2 payload can contain a list of objects derived from (in SV) `uvm_tlm_extension`. If the extension class contains only integral types, no packing/unpacking methods have to be implemented.

For the instruction fetch and load/store interfaces the payload contains read/write, address, data and byte-enables. Additional information like access-privilege level is encoded in a custom payload. For the response (read-data, status indication), we used a dedicated response extension.

The generic payload has a `response_status` field, but we used it only to report problems in the TLM2 communication itself (e.g. address greater than 32-bit, illegal byte-enables, write to read-only socket, no response received, etc.). Application-level errors are transmitted via the response extension. This separation of error semantics is highly recommended to support debugging.

G. DPI-C Interfaces

Using DPI-C, it is possible to transmit integral types and structs through the language barrier. Its typical usage is well-documented, but we want to add two examples for declaring DPI-C functions in order to highlight best-practice.

1) SystemVerilog functions that must be called from C++ are exported from SystemVerilog, e.g.
`export "DPI-C" function cExpRequestSystemReset();`

The functions are global and should be encapsulated in a package. If they have some impact on state, we recommend to simply set a package variable which can then be accessed statically from other parts of the SV environment using `<package_name>::<variable_name>`. *Tasks* can also be exported which is useful to consume time (for applications requiring blocking calls).

Import into C++ namespace:

```
extern “C” void cExpRequestSystemReset(char modelType);
```

Note that exported SV tasks may consume time. Example for exporting a task:

```
export “DPI-C” task cExpSync();
```

2) C++ functions that must be called from SystemVerilog are imported into SV, e.g. consider the C-prototype

```
int cImpInitCpu() { ... }
```

Note that the function body must set the SystemVerilog scope using `svSetScope()`.

Import into SystemVerilog:

```
import “DPI-C” context task cImpInitCpu();
```

The `context` attribute as opposed to the default `pure` attribute states that the task is accessing objects other than its input parameters (e.g. SystemC objects, exported functions) and/or accessing the SV simulation model via VPI calls.

H. Simulation Performance

Using the testbench that allows drop-in of either DUT or C-model, some performance numbers can be given. The C-model within the UVM testbench framework is 7-8 times faster than the RTL DUT. We expect better performance of the model when running stand-alone (the software simulator application was work in progress at the time of writing). This is not a comprehensive benchmark but gives a rough estimation. For verification purposes we consider performance of the C-model less important, because the SystemVerilog simulation is always the bottleneck. However, for software simulation with real-world use-cases speed is crucial.

Test	**# instructions**	**CPU time[1] RTL/s**	**CPU time[2] SysC/s**	**Instructions/s RTL**	**Instructions/s SysC**
dhrystone	3194	16,5	3,1	194	1030
pi	4409	27,6	3,1	160	1422
memory_byte_access	29569	74,9	10,9	395	2713
memory_attributes	109582	272,0	37,6	434	3138
whetstone_1	1184060	5973,0	380,0	198	3116

IV. SUMMARY

We successfully applied the UVM-ML architecture to integrate different flavors of a System-C model into a variety of UVM-SV testbenches. Some best-practices emerged from our work. The UVM-ML implementation greatly improves ease-of-use in mixed-language environments, but there is still considerable friction and overhead for the verification engineer. Ultimately, time should be spent on verifying and debugging the DUT rather than solving interoperability problems.

ACKNOWLEDGMENT

[a.] [1] *Verbosity UVM_LOW, no linedebug, no trace-file*

[b.] [2] *Verbosity UVM/SC_LOW, linedebug, no trace-file*

We would like to thank the members of both the software and verification teams who contributed to this project.

REFERENCES

[1] Cadence Help, version 15.20.

[2] "IEEE Standard for Standard SystemC Language Reference Manual", IEEE Std 1666-2011, January 2012

[3] "OSCI TLM-2.0 Language Reference Manual", Open SystemC Initiative, 2009

[4] https://en.wikipedia.org/wiki/Jenkins_(software) , Wikipedia, June 2017

[5] https://www.agilealliance.org/, Agile Alliance, 2017

[6] https://en.wikipedia.org/wiki/SystemVerilog_DPI, Wikipedia, March 2017