2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

Verilator + UVM-SystemC: a match made in heaven

Luca Sasselli, QT Technologies Ireland Limited, Cork, Ireland (*lsassell@qti.qualcomm.com*)

***Abstract* - In the last 10 years SystemVerilog and the Universal Verification Methodology (UVM) have enjoyed widespread adoption by the industry, becoming the de facto standards for digital design and verification. Despite their popularity, as of 2023 no open-source simulator supports enough SystemVerilog constructs to allow a complete implementation of UVM, relegating these methodologies to costly closed-source tools. This paper explores using Verilator, a SystemVerilog to SystemC/C++ conversion tool, and Accelera's UVM-SystemC library to build an industry level open-source verification environment. Finally, a case study is presented in which the proposed framework is used to verify a RISC-V microprocessor, taking full advantage of the features offered by this unique combination of tools and methodologies.**

Keywords - SystemC, UVM, Verilator, RISC-V

I. Introduction

Despite introducing many useful design constructs and addressing some shortcomings of Verilog, SystemVerilog owes much of its popularity to the Universal Verification Methodology (UVM)[1]. In stark contrast to its ubiquity in the industry, UVM support has historically been missing from the open-source landscape: according to the *CHIPS Alliance SystemVerilog Tester* suite[2], as of 2023 no open-source simulator is able to compile the full UVM library. This is due to the incredible undertaking of providing complete support for the extremely intricate non-synthesizable superset of the SystemVerilog language, on which UVM is built upon.

Of the existing open-source projects, Verilator represents one of the most interesting solutions. Although technically not a simulator, Verilator can convert (or "verilate") synthesizable SystemVerilog into a C++ or SystemC model that can be then compiled and executed. Its popularity has been boosted by its remarkable simulation speed: Verilator ignores IEEE 1800[3] requirement for an event-driven simulation, in favor of a cycle-based one. This makes the verilation process closer to synthesis, rather than a strict interpretation of RTL, allowing for greater flexibility in the model optimization, making simulation considerably faster than closed-source alternatives[4].

With the introduction of Accelera's UVM-SystemC[5] it is now possible to build a UVM based testbench for a SystemVerilog design using Verilator. While Verilator support for the full UVM library is planned[6], it can be argued that there are many advantages to a SystemC based testbench. Being C++ based, SystemC allows to seamlessly integrate verification components written in C/C++, including embedded software, and leverage the extensive wealth of C++ libraries. Additionally, SystemC is a popular choice for high-level system modelling and architecture exploration, and by using the UVM methodology, components developed in these phases of the design can be reused for RTL verification.

This paper explores using Verilator to verify a SystemVerilog based design with UVM-SystemC, describing how to build a testbench, providing solution for common issues and presenting a simple regression framework. These techniques are then used in a case study of RISC-V microprocessor verification, highlighting the benefits of the proposed approach. Finally, the limitations and issues found during the study are presented and discussed.

II. Related work

Due to the lack of UVM support, verification under Verilator has been traditionally reliant on "raw" C++, and among the non-UVM open-source frameworks, only the Python based cocotb[7] provides full support for the tool. Recently the excellent pyUVM[8], provided a partial implementation of UVM, with many of the base classes, factory, reporting, phasing and a simplified version of the configuration database. Although the idea of using Python for verification is alluring, there are very strong arguments against it, namely poor performance at a scale, the global interpreter lock preventing parallel execution of threads and the lack of static type checking[9].

III. Methodology

A. *Building and running regressions with CMake*

As previously mentioned, Verilator is not a simulator in the traditional sense, instead it is used to convert SystemVerilog code to a C++/SystemC model. The compilation flow is thus as follows: first RTL is verilated

into C++, then the resulting code is built with the testbench using a standard C++ compiler, and finally the resulting executable is invoked to run the simulation. To streamline this process, the tool offers a robust CMake[10] integration, in the form of the *verilate* function, a wrapper that encapsulates the RTL conversion and creates a compilation target for the resulting C++ code.

One additional advantage of using CMake is that CTest can be used to implement a simple regression framework for UVM, that natively implements many useful features such as test selection, logging, failed test replay, etc. Tests can be defined in the CMakeList.txt using the `add_test` function, specifying a name and a command to run the test; exit code is used to determine the pass or fail status. Return value can be easily obtained from UVM using the `uvm_report_server` error count, but test selection from command line must be implemented from scratch since, as of SystemC-UVM Beta 5, the library doesn't provide any command line support for UVM plus-arguments (i.e. `uvm_cmdline_processor`). An example of testbench *main* function is as follows:

```
#include <boost/program_options.hpp>

using namespace boost::program_options;

int main(int argc, char** argv) {

  options_description desc{"Options"};
  desc.add_options()
    ("uvm_testname", value<string>()->required(), "UVM test name");

  variables_map opts;
  store(parse_command_line(argc, argv, desc), opts);

  ...

  uvm::run_test(opts["uvm_testname"].as<string>());

  if(uvm::uvm_report_server::get_server()->get_severity_count(uvm::UVM_ERROR) > 0) {
    return 1;
  }

  return 0;
}
```

In this example, `boost::options` is used for command line parsing, forgoing the SystemVerilog plus-argument in favor of a UNIX-like syntax. To determine the exit code only the *UVM_ERROR* count is required, since *UVM_FATAL* already results in a `std::runtime_error`.

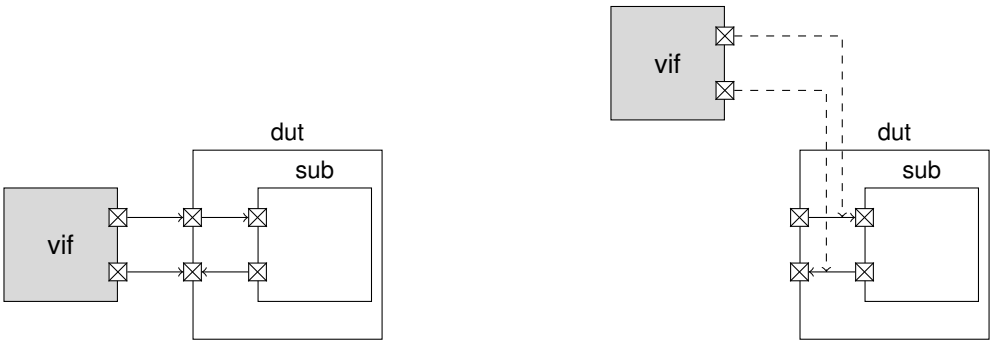
A CMakeList.txt for a Verilator based UVM-SystemC project can be as follows:

```
cmake_minimum_required(VERSION 3.12)

project(dvcon2023)
find_package(verilator HINTS $ENV{VERILATOR_ROOT} ${VERILATOR_ROOT})
find_package(Boost REQUIRED COMPONENTS program_options)

# Design
add_library(dut)
verilate(dut
  SYSTEMC TRACE
  SOURCES dut.sv)

verilator_link_systemc(dut)

# Testbench
add_executable(sim
  testbench.cpp)
include_directories(${Boost_INCLUDE_DIRS})
target_link_libraries(sim PRIVATE dut)
target_link_libraries(sim PRIVATE uvm-systemc)
target_link_libraries(sim PRIVATE ${Boost_LIBRARIES})

# Tests
enable_testing()
```

```
add_test(
  NAME "test1"
  COMMAND sim --uvm_testname sanity)

...
```

RTL is verilated and compiled in a separate library that is then linked to the simulator executable, isolating verilation from C++ code: the rest of the CMake configuration can be treated as a standard C++ project. Boost library is also added to allow using `boost::options`. In the last portion of the example, a sample test, called *test1* based on UVM test *sanity*, is defined for CTest: this and other tests defined can be executed by calling the *ctest* executable.

B. *Connecting VIPs to a verilated DUT*



(a) Boundary connection (b) Hierarchical connection

Figure 1: Agent connections

UVM-SystemC handles connecting VIPs to the DUT very similarly to its SystemVerilog counterpart: an interface, in form of a sc_module, is connected to the DUT instance and registered in the uvm_config_db; this interface is then retrieved by an agent and used to interact with the design.

```
sc_module(vip_if){

  sc_signal<...> foo;
  sc_signal<...> bar;

  SC_CTOR(vip_if){}
};

int main(int argc, char** argv) {

  ...

  vip_if* vif = new vip_if("vif");
  uvm_config_db<vip_if*>::set(uvm_root::get(), "*", "vif", vif);
  dut->foo(vif->foo);
  dut->bar(vif->bar);

  ...
}
```

When connecting to a verilated model primary IOs (figure 1a), this approach can be followed verbatim, but some care must be taken when dealing with port types. To improve simulation performance, Verilator handles as many signals as possible with native C++ types, i.e. single bit logics are cast to *bool*, buses 32 bits or smaller to *uint32_t*, buses 64 bits or smaller to *uint64_t* and only larger buses to *sc_bv*. These changes are reflected on the verilated sc_module IOs and must be taken into account when creating the connections.

Unfortunately, connecting to internal signals (figure 1b) is not as straight forward as a SystemVerilog hierarchical reference: for performance reasons Verilator doesn't generate all internals as SystemC, in fact, only a wrapper sc_module is created around a pure C++ model of the design. Additionally, it's possible that the signals we want to connect to are optimized by the verilation process, and thus might not exist at all in the model. To solve this issue, Verilator allows preserving and exposing signals of interest to the testbench, either by specifying them at verilation time, or by marking them in RTL using metacomments like `/* verilator public */`. Exported signals however are pure C++ variables, thus can't be connected directly to the SystemC testbench. While it is possible to read the exported variables into the interface sc_signals from a SystemC process, using either a fast clock or the system clock, this can lead to race conditions when some signals probed are driven by the TB directly. Instead, a better solution is to use the *Simulation Phase Callbacks* introduced by SystemC 2.3.1, to read the signals every time the simulation kernel executes the *Advance Time* step:

```
template<typename T> vif_internal_if: public vip_if
{
  T* ptr;

  vip_internal_if(const sc_module_name& name) : vip_if(name) {
    register_simulation_phase_callback(SC_BEFORE_TIMESTEP);
  }

  void simulation_phase_callback() {
    foo = ptr->foo;
    bar = ptr->bar;
  }
};

int main(int argc, char** argv) {

  ...

  vip_internal_if* vif = new vip_internal_if("vif");
  uvm_config_db<vip_if*>::set(uvm_root::get(), "*", "vif", vif);
  vif->ptr = dut->sub;

  ...
}
```

C. *Dumping waveforms*

Verilator allows dumping waveforms in VCD format by passing the `-trace` switch to the tool, or the `TRACE` argument to CMake: this enables waveforms generation is the verilated model. Tracing can be then enabled by passing an instance of this object to the DUT using the `trace` method. This must be done after the SystemC elaboration phase has completed, to allow the simulation back-end to create all the data structures to be dumped.

```
#include "verilated_vcd_sc.h"

...

int main(int argc, char** argv) {

  ...

  verilated::traceEverOn(true);
  verilatedVcdSc* tfp = new verilatedVcdSc;

  sc_start(SC_ZERO_TIME); // Trigger SC elaboration phase
  dut->trace(tfp, 99); // Trace 99 levels of hierarchy
  tfp->open("wave.vcd");

  ...

  sc_start(...); // Run the simulation
```

```
  ...

  tfp->close();
}
```

This unfortunately can't be done while using UVM-SystemC, since control of the simulation is handled by the UVM environment, and `sc_core::sc_start` is called by `uvm::run_test`. A possible solution is to create a dummy sc_module and override the `start_of_simulation` and `end_of_simulation` callbacks, to begin and end the waveform tracing. This however can cause incomplete waveform dumps if the UVM test is terminated by a `std::runtime_error`, such in the case of when `uvm::die()` is called (e.g. timeout, UVM_FATAL, etc.). Instead, a dummy uvm_component can be created to allow access not only to the simulation phases callbacks, but also to `pre_abort`:

```
template<typename T> class uvm_trace : public uvm_component {
 public:
  UVM_COMPONENT_PARAM_UTILS(uvm_trace);

  uvm_trace( uvm::uvm_component_name name, T* obj)
    : uvm_scoreboard(name),
      obj(obj) {
    verilated::traceEverOn(true);
    verilatedVcdSc* tfp = new verilatedVcdSc;
  }

  void start_of_simulation_phase(uvm_phase& phase) { start(); }
  void end_of_simulation_phase(uvm_phase& phase) { stop(); }
  void pre_abort() { stop(); }

 private:
  verilatedVcdSc* tfp;

  void start(){
    obj->trace(tfp, 99);
    tfp->open("wave.vcd");
  }

  void stop(){
    tfp->flush(); // Ensures that all pending data is written
    tfp->close();
  }
}

int main(int argc, char** argv) {

  ...

  uvm_trace<DutType> trace = new {"trace", dut};
  run_test(...);

  ...
}
```

IV. Case Study

The techniques described in III. were used to verify a simple RISC-V processor written in SystemVerilog. The DUT is a traditional 5-stage pipeline architecture implementing the RV32IM instruction set, with a Wishbone crossbar to allow shared access to the memories and to implement a dedicated peripheral port. This design was chosen exclusively to be an exercise and doesn't intend to be particularly optimized for any real-life application.

The testbench, shown in figure 2, is organized as follows: three Wishbone agents are connected to the instruction, data and peripheral bus slave ports, and two master agents are connected to the bus master ports. A dedicated pipeline agent is used to monitor the pipeline state to collect the program counter, instruction fetched, register file and memory accesses for each instruction executed.

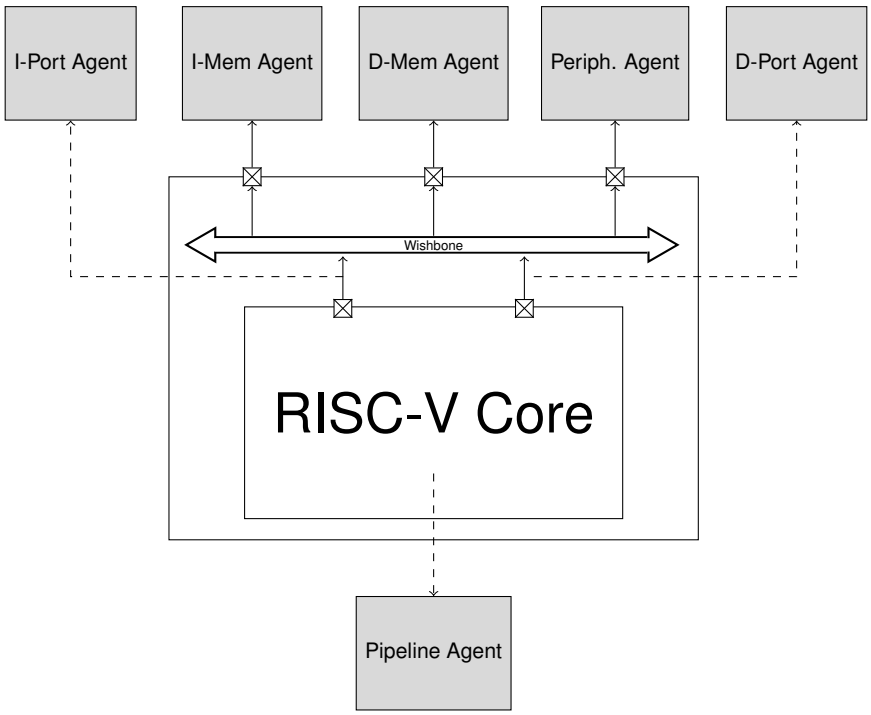


Figure 2: Testbench architecture

The RISC-V core was verified using the *SPIKE RISC-V ISA simulator*[11] as golden reference, by comparing the pipeline activity captured by the pipeline monitor, with the one produced by the tool. The *riscv-test*[12] suite was used as stimuli for the core, since it provides a comprehensive official self-checking set of unit tests for RISC-V cores. To simplify the verification environment the Spike executable was invoked directly within the SystemC scoreboard using `boost::process`, and the reference transactions were extracted directly from stdout using `boost::regex`. While a similar approach is possible in a SystemVerilog based testbench using `\$system`, this runs the executable synchronously when the function is called, whereas C++ allows to run it concurrently to test execution in a separate thread, with bidirectional communication.

During core tests, instruction and memory agents run a simple memory model sequence, preloaded respectively with the code and data segments of the firmware. These can be extracted directly from the .elf file within C++ using libelf, along with other useful symbols, such as the reset address and the memory location of `to_host`/`from_host`, used by Spike to implement test to host communication for test status and log. This allows to dynamically obtain the configuration required to run a generic firmware, without the need of using a testbench specific linker script.

Bus verification was instead performed with traditional contrained-random verification, using a model of the crossbar and a scoreboard to check the RTL behavior. In RISC-V based tests, the instruction and data port agents were used as passive monitors, while in dedicated bus test acted as active masters, using the `/*verilator forceable*/` meta-comment to take direct control of the internal signals from the testbench.

V. Limitations

A. *Verilator*

Verilator's cycle-based simulation is both a curse and a blessing: on one hand, it enables very fast simulations, but on the other, all intra-cycle information is lost. This means that for timing simulations, back-annotated netlist and clock-less logic another tool is needed. Additionally, Verilator support for concurrent assertions and coverage is still partial, limiting their use for verification. Finally, the conversion from SystemVerilog to C++ severely limits the reuse of design constructs (like interfaces, structs, types, and parameters) in the testbench, leading to unnecessary duplication.

B. *UVM-SystemC*

UVM-SystemC is currently in beta status and exists mostly as a proof-of-concept: some components are still partially implemented, missing or received limited testing. Nonetheless, this paper's experiments highlighted no major issue with the Beta 5 version of the library. Transitioning for SystemVerilog UVM is relatively straightforward, but requires a robust knowledge of C++ and at least basic knowledge of SystemC, mostly to deal with data types, synchronous statements and threads.

C. *Coverage*

Although Verilator supports both code and functional coverage, the current implementation is somewhat limited. Only code and toggle metrics are available for code coverage, and the former is compressed across modules with identical parametrization, making instance based analysis impossible. Functional coverage only supports the assertion syntax (i.e. *cover property*) and more advanced statements such as *covergroups* are not yet supported. On the SystemC front, libraries such as FC4SC [13] can provide rudimentary SystemVerilog-like functional coverage.

While this might sound sufficient for some applications, there's the issue of how to handle the coverage data. Verilator uses a custom format, but allows conversion to lcov *.info* format (a popular gcc's gcov front-end) allowing to generate HTML report of the coverage. The process however is lossy: since lcov was created for software coverage all results are reported as a line score, making the results hard to interpret. FC4SC on the other end uses the Accelera's Unified Coverage Interoperability Standard (UCIS)[14], a format that unfortunately has seen limited support by both vendors and open-source community.

D. *Random stability*

Random stability is an important property of the SystemVerilog language, and a critical aspect of constrained random-verification. During the design cycle the existing test library is periodically regressed with different seeds, to ensure an exhaustive exploration of the design state space. If an error is highlighted by a test, random stability guarantees that the same seed will produce the same exact stimuli that caused the error, and if the design or testbench have changed, minimizes the impact on the randomized values. This is achieved by using a random number generator (RNG) unique to each thread and object, to ensure that the random number generation is independent of the order of the calls across them.

While Verilator is fully complaint to the SystemVerilog random stability requirements, SystemC lacks any such feature, and existing constrained-random frameworks, like CRAVE[15], rely on a global RNG: this makes the testbench randomization extremely sensitive to code changes and the thread scheduling order.

VI. Conclusion

In conclusion, Verilator and UVM-SystemC are a feasible open-source solution for verifying a SystemVerilog design of the complexity of the case study. Verilator provides a very robust solution for RTL simulation and, despite the limitations described in A., requires little adjustment on the design side. UVM-SystemC is an excellent choice for verification, allowing to use the UVM alongside many powerful C++ libraries, but it requires a robust C++ knowledge to transition from SystemVerilog. The open nature of these tools drastically simplifies debugging and addressing any issue with them, such as the ones described in III..

References

[1] "IEEE Standard for Universal Verification Methodology Language Reference Manual". In: *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)* (2020), pp. 1–458. DOI: 10.1109/IEEESTD.2020.9195920.

[2] CHIPS Alliance. *SystemVerilog Tester: Test suite designed to check compliance with the SystemVerilog standard.* URL: https://github.com/chipsalliance/sv-tests.

[3] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[4] Schuyler Eldridge. *Simulation speed for rocket-chip: VCS vs Verilator*. URL: https://github.com/chipsalliance/rocket-chip/issues/2160.

[5] Accelera. *UVM-SystemC Public Review*. URL: https://accellera.org/itc/hm/0209.html.

[6] antmicro. *Dynamic scheduling in Verilator - milestone towards open source UVM*. URL: https://antmicro.com/blog/2021/05/dynamic-scheduling-in-verilator/.

[7] FOSSi Foundation. *cocotb is an open source coroutine-based cosimulation testbench environment for verifying VHDL and SystemVerilog RTL using Python.* URL: https://www.cocotb.org/.

[8] Lars Asplund. *pyUVM*. URL: https://github.com/pyuvm/pyuvm.

[9] Rich Porter. *Why you shouldn't use Python for Verification*. URL: https://www.youtube.com/watch?v=KoE-Jp6OFwI.

[10] Kitware. *CMake*. URL: https://cmake.org/.

[11] RISC-V International. *Spike RISC-V ISA Simulator*. URL: https://github.com/riscv-software-src/riscv-isa-sim.

[12] RISC-V International. *riscv-tests*. URL: https://github.com/riscv-software-src/riscv-test.

[13] Amiq-Consulting. *FC4SC: A header only C++11 library for functional coverage*. URL: https://github.com/amiq-consulting/fc4sc.

[14] Accelera. *Unified Coverage Interoperability Standard*. URL: https://www.accellera.org/activities/working-groups/ucis.

[15] University of Bremen. *CRAVE - Constrained Random Verification Environment*. URL: https://github.com/agra-uni-bremen/crave.