2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

A Framework for the Execution of Python Tests in SystemC and Specman Testbenches

Christoph Tietz[1], Sebastian Stieber[2], Najdet Charaf[3], Diana Göhringer[3]
[1] Bosch Sensortec GmbH, Dresden, Germany (christoph.tietz@bosch-sensortec.com)
[2] StZ System-Level-Modellierung, Kritzmow, Germany (SU2297@stw.de)
[3] Technische Universität Dresden, Chair of Adaptive Dynamic Systems, Germany
({najdet.charaf|diana.goehringer}@tu-dresden.de)

***Abstract* — Modern HW/SW co-design approaches require the parallel development of hardware (HW) and software (SW) in order to meet demanding time-to-market goals. That, in turn, results in both HW and SW development teams working with different platforms and tools, creating a gap between the two domains and making reuse difficult. The SW team usually develops and tests its components with a virtual prototype implemented in SystemC, for example. For an early availability of this model, the abstraction level is chosen higher and results in differences compared to the real HW. Therefore, it is necessary to validate the functionality of the SW components also in a more precise platform, e.g. RTL simulation, which is typically used by the HW development team and can be carried out in Specman testbenches. In this work, we present a solution to close the gap between top-level SystemC and RTL simulations. We propose a framework that allows executing the same test cases, implemented in Python, in both simulation platforms without additional effort. Thus, the SW development team has easy and flexible access to SystemC and RTL simulation. For this purpose, we have implemented a Python API for SystemC and Specman testbenches. The API facilitates scripted host interaction with the device under test (DUT) and simulation control.**

Keywords — Python; SystemC; Specman; HW/SW co-design; horizontal reuse

I. INTRODUCTION

Challenging product requirements and tighter time-to-market constraints constantly increase the project risks of ASIC designs. In order to handle this complexity and reduce the risk, hardware/software (HW/SW) co-design is applied. In recent years, many approaches have been developed to optimize this process. While the software was once implemented sequentially after the hardware, it is now common practice to develop the software in parallel to the hardware using a virtual prototype [1]. Such a model, e.g. implemented in SystemC, allows software developers to test their components early in the development process with great debug capabilities and fast simulation run-times. Directed testing is often sufficient for small-sized software without any functional safety requirements. On the other side, the hardware is fixed after tape-out; hence, functional correctness is crucial to avoid cost explosions. Sophisticated methodologies and hardware verification languages like UVM, OVM or Specman e have been developed to enable an efficient verification process in RTL simulation.

Different tools and simulation platforms of the distributed HW and SW teams hinder effective information exchange and bring new challenges to the development process. A high abstraction level of SystemC virtual prototype facilitates early availability in the development process, thus allowing a fast ramp-up of the SW development. However, this abstraction also creates discrepancies to the actual behaviour of the hardware, e.g. in terms of timing, accuracy and functionality. In order to detect functional bugs before tape-out, more accurate platforms like FPGA prototypes, RTL simulation or emulators are required. While prototyping needs further design effort and has limited observability of signals, emulators are very costly and hence, only limited available for developers. Consequently, the RTL simulation is the most suitable platform for testing small HW/SW co-designs. Horizontal reuse of existing SW test resources from the virtual to the RTL platform is necessary to be competitive with the time-to-market constraints. In recent years, advanced methodologies, e.g. the Portable Test and Stimulus Standard (PSS) [2], popped up addressing this reuse issue across multiple platforms. In our work, we present a solution that allows reuse of tests across SystemC and RTL top-level simulations without any further required tool costs and language knowledge. For this purpose, we encapsulate the test description in Python tests, also referred to as test cases in the following, that can interact with the DUT host and simulation environment with a specific set of API methods. The proposed framework implements the necessary APIs to the SystemC and Specman testbenches.

In hardware development, with sophisticated testbenches, the verification team cares about maintaining the simulation environment. Since the verification process is already very time-consuming [3], the additional interface should not cause any further overhead. Therefore, the Python Test - Specman interface is designed as a generic package to be reusable across projects with minimized integration effort.

The remainder of the paper is organized as follows: State of the art of embedding scripts to SystemC and RTL environments is presented in the next section. Then, the design and implementation of each API of the framework is described. At the end, results and the conclusion will be shown.

II. State of the Art

Enabling scripting inside EDA tools already has a long history [4]. However, the purpose and use of the scripting language can be quite different. [5] lists possible fields of applications: automation, configuration of simulation patterns, assembling the simulation, extending the simulation with high-level models, test integration and interactive introspection. This work focuses on test integration. More specifically, the scripting language should be used as a test description for SystemC and RTL simulations to gain efficiency in testing and reusing test stimuli. A similar intention is pursued in [6]. The authors investigate various approaches to connect Perl and SystemC to use abstract test descriptions for improved efficiency in pre-silicon design verification and reuse of test stimuli in post-silicon platforms. In their case study, code generation, code conversion, code transformation and the embedded approach are examined in terms of various metrics. Although the simulation runtime is not the best, the authors conclude that embedding the scripting language is the most suitable concept since it requires no test adaptions. In their proposed pre-silicon verification approach, the SystemC environment includes, besides the instruction-set simulator and application code, also the hardware design for an HW/SW co-simulation. In our work, however, the intended interface to the RTL simulation targets the Specman testbench to reuse the existing environment, including checks and coverage of hardware design flows with Specman. Furthermore, the proposed package for Specman testbenches is reusable across ASIC designs that use one host interface (HIF) for communication.

Integrating Python into RTL simulation can be achieved with Python libraries like cocotb [7] allowing interaction with simulators over standardized interfaces. However, in design flows where UVM or Specman is already used, cocotb would be hard to integrate. The hardware verification languages also offer other opportunities to connect different languages. In [8] is shown how SystemVerilog or Specman e can be connected to Python using the direct programming interface C (DPI-C). While this approach requires some additional engineering effort, Cadence recently introduced direct support of Python within Specman [9]. The proposed Specman API uses this built-in interface to facilitate Python code execution and data exchange.

III. Overview of the proposed Framework

A high-level perspective of the framework usage is shown in figure 1. In our HW/SW co-design flow, the framework targets easy access to the SystemC simulation platform for SW developers to test their components. For this purpose, Python scripts are used for test descriptions. As soon as RTL is available in the project flow, the framework's Specman API enables the reuse of these tests in RTL simulation. The following sections describe the design concepts of the framework's interfaces to both simulation platforms. Furthermore, implementation details for one exemplary interface method are shown.

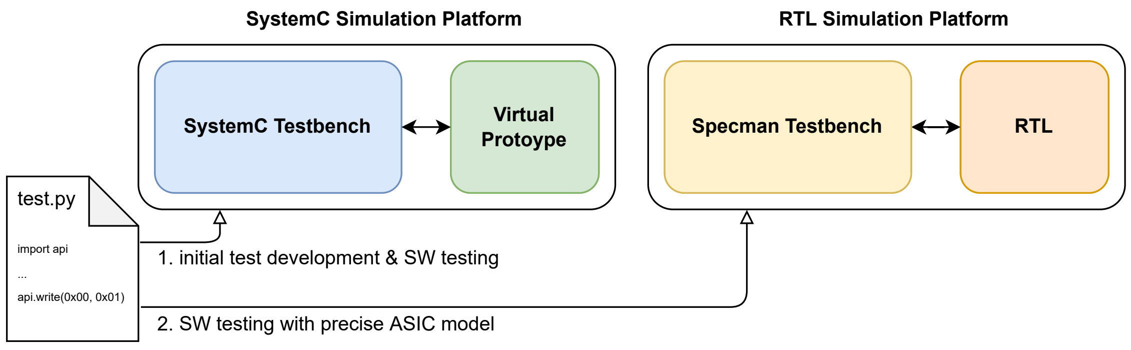

Figure 1: The framework facilitates the execution and reuse of Python tests in SystemC (1) and RTL (2) simulation

A. Python Test - SystemC Model API in Detail

A virtual prototype of an embedded system requires a testbench to define the stimuli of a simulation run and to gather simulation results. The testbench of a SystemC model is often implemented in C++ and already defined at compile time. However, offloading the test description to a scripting interface makes the virtual prototype much more flexible. Adapting the stimuli and automating the test flow becomes much handier, and the usage of the virtual prototype as a regression platform gets simplified. Basically, there are a lot of embeddable scripting languages like Python, Tcl, Lua and much more feasible for this task. Especially Python is a very popular and powerful scripting language that supports a variety of additional libraries to process and visualize simulation data directly at simulation run-time. The extensive documentation and large community also lower the entry barrier for users of the virtual prototype. Hence, our C++/SystemC virtual prototype embeds a Python3 interpreter to allow a scripting-based simulation control. The high-level architecture is shown in figure 2.

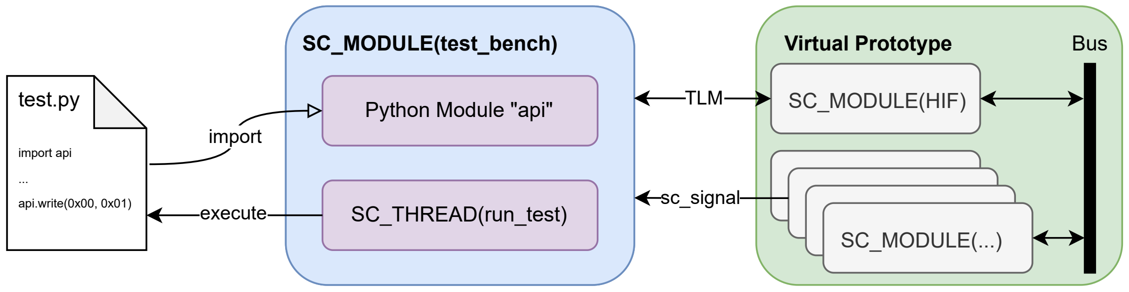

Figure 2: Overview of the SystemC Simulation Platform

The interpreter is embedded in a SystemC testbench module, which is connected to the other modules of the virtual prototype using standard interfaces like ports, signals and TLM simple sockets. A set of predefined C++ functions is bound to a Python module, creating a predefined interface *"api"*. The script must import this module in order to get access to these interface methods. After elaboration, the test script is loaded and executed within a SC_THREAD process allowing the control of the module interfaces and progress of simulation time. It can generate events at simulation run-time based on the contents of the test script. Besides other works, transactions on the prototype's serial communication interface can be generated. It also enables some debug functionality like reading or writing to simulated internal memory blocks. Furthermore, incoming events can be used to call Python methods asynchronously after they have been registered as callback methods. Table 1 lists a subset of implemented API methods that can be used in the test script to communicate with the virtual prototype.

Table 1: Python API Methods for both Simulation Platforms

Method Name	**Description**
wait(time, unit)	Let the simulation proceed for the given time.
wait_until(time_in_s)	Let the simulation proceed until the given absolute simulation time in seconds.
read(address, length)	Reads the given number of words from selected register address.
write(address, [list of data])	Writes the given list of data words in burst mode starting at the selected register address.
read_dbg(address, length)	Reads the given number of words from selected register address without consuming time or issuing logical side effects like clear on read.
read_ram(address, length)	Reads the given number of words from given memory address without consuming time.
write_ram(address, [list of data])	Writes the given list of data starting at the selected ram address without consuming time.
register_cb_int*(callback)	Sets a callback function to get notified for interrupt pin * level changes.
wait_int*_pos|neg(timeout_in_ms)	Let the simulation proceed until the next positive|negative edge of interrupt pin * occurs. Optionally, a timeout value (in milliseconds) can be given.
log(message)	Prints a message in combination with the simulation timestamp.
get_st()	Returns the current simulation time in seconds.
stop(error_message)	Immediately stops the simulation. Optionally, prints an error message.

The binding of the C++ methods to the Python API module shall be shown exemplary for the `api.read(addr, length)` call. This method can be used to read one or more words from the serial interface of the virtual prototype. This communication is implemented using a TLM simple socket interface sending generic payloads in a blocking manner. Hence, simulation time can advance while this communication is ongoing. Inside the testbench module, a static function needs to be added which parses the Python arguments, sends the TLM transaction and generates the Python return values. Listing 1 shows such an exemplary implementation. A singleton is used to provide access to the class member TLM socket. This function must be added to a method table and later to the Python API module in order to make it available to the test script as shown in listing 2. Inside the SC_THREAD process of the testbench module, the Python interpreter gets initialized and the API module is imported. Finally, the test script is parsed by the interpreter. The execution of the simulation will stay inside of the testbench module until an API method is used that pauses the SC_THREAD process, e.g. by calling `sc_core::wait`.

Listing 1: read API method implementation in SystemC

```
static PyObject *read_cmd(PyObject *self,
    PyObject *args) {
  uint8_t addr;
  uint16_t len;

  // parse python arguments
  if (!PyArg_ParseTuple(args, "BH", &addr, &
    len)) return NULL;
  // create buffer for read results
  uint16_t *data = new uint16_t[len];

  // send TLM transaction
  // access to testbench module via singleton
  TB_IF::get_instance()->get_TB()->read_tlm(
    data, addr, len);
  // create python list from result buffer
  PyObject* list = PyList_New(len);
  for (uint16_t i = 0; i < len; i++) {
    PyList_SetItem(list, i, Py_BuildValue("H"
    , data[i]));
  }

  delete[] data;
  return list;
}
```

Listing 2: Registering API method

```
// create method table
static PyMethodDef method_table[] = {
  {"read", read_cmd, METH_VARARGS, "Read"},
  {NULL, NULL, 0, NULL}
};
// create api module
static struct PyModuleDef apimodule = {
  PyModuleDef_HEAD_INIT, "api", NULL, -1,
    method_table
};
// bind api module
PyMODINIT_FUNC
PyInit_api(void) {
  return PyModule_Create(&apimodule);
}
// SC_PROCESS
void TB::run_test() {
  // initialize python interpreter
  PyImport_AppendInittab("api", PyInit_api);
  Py_InitializeEx(0);
  // open and execute test script
  FILE *fp = _Py_fopen(script.c_str(),"rb");
  PyRun_SimpleFileEx(fp, script.c_str(),1);
  Py_Finalize();
}
```

B. *Python Test - Specman API in Detail*

In our framework, the same python interface methods, as listed in table 1, are also provided to the RTL simulation platform in order to reuse the tests from the virtual platform. For a minimum setup and integration effort into existing Specman testbenches, the interface is provided as a generic package, namely Python Test Case (PyTC) package. Its main components and usage within a simplified RTL top-level testbench are shown in Figure 3.

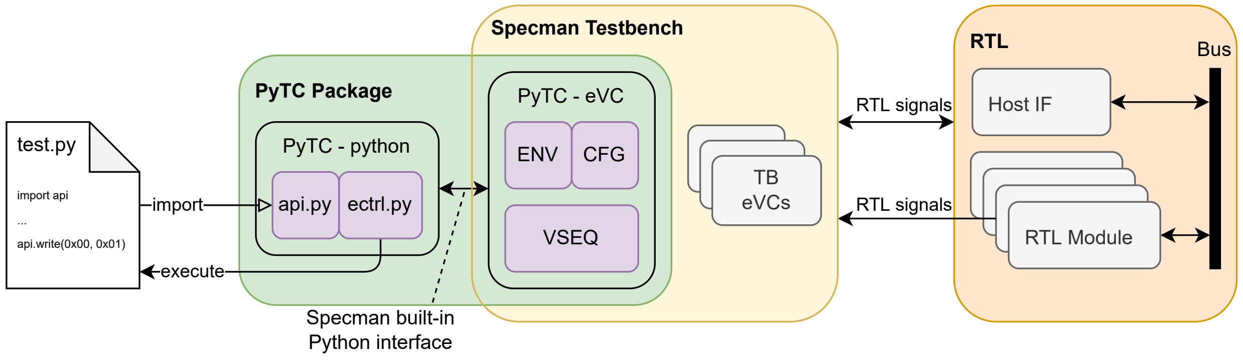

Figure 3: Overview of the RTL testbench

It includes Python modules and an e Verification Component (eVC). The communication between these components is realized with the Specman built-in Python interface that allows the execution of Python methods from Specman code with a user-defined interpreter. The eVC's environment (ENV) provides such method hooks to *ectrl.py*, facilitating Python test case execution and communication across this test and the Specman testbench. The configuration (CFG) provides debug switches and a field containing the path of the Python test. This implies that each Python test to be reused must have an equivalent Specman test that constrains the testbench appropriately. After the generation phase of a Specman test, the defined Python test case is executed in a new Python thread. To transfer the API method calls to Specman, the name and arguments are packed into data structs and transferred through global queues. For this purpose, the *api.py* defines the appropriate methods imported by the test. Handling of the communication queues is implemented in *ectrl.py* providing an API command and response channels to the Python and Specman domain. The PyTC sequence of the virtual sequence driver (VSEQ) uses these channels to fetch API method calls, evaluates them and sends the responses back to the Python test. Furthermore, it also allows the registration of asynchronous Python callbacks that are handled on separate communication channels.

For integration in existing RTL top-level simulation environments, the package only needs to be imported and a few defines and callback events for the VSEQ must be configured. For other API methods that are not part of the package, e.g. project-dependent extensions, the interface can be easily extended using hooks in the eVC.

The implementation of the read method in *api.py* is shown in listing 3. The command is passed to Specman using the *ectrl.py* utility function `process_api_cmd(...)` that also blocks the Python test case execution until the eVC answers the request. On the Specman side, this request is handled in the VSEQ's sequence as shown in listing 4. It fetches the command, processes it and returns the response back to the Python test. For a whole Python test this procedure is looped until the test ends.

Listing 3: Read method's implementation in *api.py*

```python
def read(reg_addr, length):
  # add new object to API command queue and return value from the response queue
  # Python test execution will be blocked until Specman/the DUT responses
  response = ectrl.process_api_cmd('read', reg_addr = reg_addr, read_length = length)
  return response.read_data
```

Listing 4: VSEQ's sequence body including implementation of the API read method

```
// sequence and VSEQ name are adpated with generics to the testbench
extend BST_PYTC_PYTHON_TC BST_PYTC_VSEQ {
  // sequence body
  body()@driver.clock is {
    var api_cmd              : bst_pytc_api_cmd_s = NULL; // current API command
    var api_cmd_response     : bst_pytc_api_cmd_response_s; // current response
    var python_exec_end      : bool; // Python test case end flag
    // process API commands until test case ends
    while (not python_exec_end) {
      api_cmd = get_api_cmd(); // fetch API command from Python queue
      api_cmd_response = new;
      // switch case for different API commands
      case (api_cmd.cmd) {
        BST_PYTC_API_CMD_READ : {
          api_cmd_response.set_read_data(read_mem(api_cmd.reg_addr, api_cmd.read_length));
        }; // ...
      };
      // write to Python response queue
      push_api_cmd_response(api_cmd_response);
    };
  };
  // memory operation for write_mem/read_mem
  !bst_pytc_mem_op : MEM'op_mode vr_ad_operation;
  // execute read operation on testbench RSD
  read_mem(addr : uint, data_len : uint) : list of byte @driver.clock is {
    do bst_pytc_mem_op on driver.BST_PYTC_RSD keeping {
      .direction     == READ;
      .address       == addr;
      .num_of_bytes == data_len * BST_PYTC_WORD_SIZE;
    };
    result = bst_pytc_mem_op.data;
  };
  // ...
};
```

IV. Results

The presented SystemC API targets an easy test interface to the virtual prototype by providing selected methods to the user. Embedding the Python interpreter into the virtual prototype facilitates a flexible and agile development platform since tests can be changed and rerun without re-compilation of the SystemC model. The Python Test - Specman API package, which is built up on the integrated Python interface, was successfully validated in the development environments of a few ASIC projects for inertial measurement units. It has shown similar advantages regarding re-compilation. Once RTL and Specman are compiled, the Python interface still allows changes on the test case, which are visible after rerunning the simulation, gaining more interactivity also in RTL simulation. First approaches to transfer the same test cases from SystemC in RTL simulation included automatic and manual translations from Python to Specman tests. While manual translation is inefficient, automatic translation had limited applicability since not all Python language constructs were supported. With the proposed Specman API, there are no longer such restrictions since a selected interpreter naturally executes the Python code.
The Python Test - Specman API package was designed and implemented to keep the integration effort into the RTL simulation platform low. Since only a few parameters and environment adaptions must be done, the setup time is less than one hour. The utility functions in ectrl.py and method hooks in the eVC also provide great extensibility, e.g. for project-dependent API methods, minimizing the implementation effort. Although the package uses a high-level interface from Specman to Python, the negative impact on simulation performance in an RTL environment is negligible, as experiments have shown. In order to determine this impact, equivalent Python and pure Specman tests were simulated in the same RTL environment. On average, the Python tests with 100 and 1000 write and read operations ran 0.4% and 2% longer, respectively.
Another approach for executing the same test on both platforms would be to entirely replace Python with Specman tests since SystemC can also be simulated with Specman. In this case, the Specman tests could be directly reused in the RTL simulation. In practice, however, this is not beneficial overall. The simulation with Specman requires licenses and is not platform-independent regarding the host machine. Furthermore, Specman can only simulate SystemC with a specific library version and compiler, limiting the creation of the virtual prototype. In comparison, using the SystemC model with our proposed Python API is free from license costs, platform-independent and more common to engineers. Experiments have also shown that the SystemC simulation with Python is much faster than SystemC in Specman by a factor of 5 to 20 for equivalent test cases.

V. Conclusions

This paper presented two APIs allowing the execution of the same Python test in SystemC and RTL simulation. The API is defined by a specific set of methods for scripted host interaction and simulation control that are supported in both simulation platforms. The usage of both APIs enables an efficient SW testing methodology where the SystemC simulation is used first and the more precise RTL simulation for final confidence later in the project flow.

The API for RTL simulation is realised as a generic Specman package to keep the integration effort for different projects low. Its functionality has been successfully tested in some ongoing ASIC projects. We showed that the impact on the simulation runtime is small, even though the package uses the built-in high-level interface from Specman to Python. The evaluation of the reuse of test cases from SystemC to RTL for a whole project is still pending. However, it is expected that all tests that use the supported API methods of both platforms and rely on the same testbench constraints can be directly reused.

Acknowledgment

The authors would like to thank Robert Richter (Bosch Sensortec GmbH) for his support in designing and testing of the PyTC package. We also thank Thomas Klotz (Bosch Sensortec GmbH) for comments on this paper.

References

[1] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL design and verification. A prescription for electronic system-level methodology*. Elsevier/Morgan Kaufmann, 2007. ISBN: 9780080488837.

[2] Accellera. “Portable Test and Stimulus Standard - Version 2.0”. In: (2021).

[3] Harry Foster. *2020 Wilson Research Group functional verification study - IC/ASIC functional verification trend report*. Tech. rep. Wilson Research Group, 2020.

[4] Pinhong Chen, D.A. Kirkpatrick, and K. Keutzer. “Scripting for EDA tools: a case study”. In: *Proceedings of the IEEE 2001. 2nd International Symposium on Quality Electronic Design*. 2001, pp. 87–93.

[5] R. Meyer, J. Wagner, R. Buchty, and M. Berekovic. “Universal scripting interface for systemc”. In: *DVCon Europe Conference Proceedings*. 2015.

[6] Dominik Widhalm, Stefan Tauner, and Martin Horauer. “Augmenting pre-silicon simulation by embedding a scripting language in a SystemC environment”. In: *2016 12th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*. 2016, pp. 1–6.

[7] cocotb. URL: `https://docs.cocotb.org/en/stable/index.html` (visited on 05/19/2022).

[8] AMIQ. *How to Connect SystemVerilog with Python*. URL: `https://www.amiq.com/consulting/2019/03/22/how-to-connect-systemverilog-with-python/` (visited on 06/16/2022).

[9] Cadence. *Specman: Python Is here!* URL: `https://community.cadence.com/cadence_blogs_8/b/fv/posts/specman-python` (visited on 05/19/2022).