2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Universal Scripting Interface for SystemC

Rolf Meyer, E.I.S., TU Braunschweig, D-38106 Braunschweig, Germany (meyer@c3e.cs.tu-bs.de)

Jan Wagner, E.I.S., TU Braunschweig, D-38106 Braunschweig, Germany (wagner@c3e.cs.tu-bs.de)

Rainer Buchty, E.I.S., TU Braunschweig, D-38106 Braunschweig, Germany (buchty@c3e.cs.tu-bs.de)

Mladen Berekovic, E.I.S., TU Braunschweig, D-38106 Braunschweig, Germany (berekovic@c3e.cs.tu-bs.de)

***Abstract*–The integration of scripting languages such as Tcl, Perl or Python in EDA tools is a well-known practice [1]. This is typically done using vendor-specific APIs featuring vendor-chosen languages. Such solutions are typically as general as possible, and therefore not always the best-suitable solution to a user's problem solving. In order to mitigate this problem, this paper presents a Universal Scripting Interface (USI) for SystemC platforms. With USI, we propose an API based on SWIG, which overcomes the aforementioned shortcomings by connecting the APIs of SystemC standard models with a scripting language. USI therefore makes all scripting language capabilities available in the targeted simulation engine and most important of all, it allows operating third party tools via a uniform abstraction on API level. For this, we introduce a language-agnostic interface, standardizing the access to standard models but not restricting the user to a certain language.**

Keywords—SystemC, Accellera, QuestaSim, TLM, Python, Tcl, scripting

1. Introduction

Scripting plays an important role in the current chip-design methodology: it allows design-flow automation, integrating a variety of tools, patching designs in the middle of the flow, extraction of required information, modifying an output format, or preparing a configuration file. Moreover, scripting languages are also used extensively within several EDA tools for providing basic programming capabilities, such as defining variables, defining option values, executing and recording command sequences, on/off switching of features, capturing output results, branching and looping, importing and exporting options, and bridging the gaps to other abstraction levels [1].

To enable automation of time-consuming and complex jobs, EDA tools come with various scripting interfaces with most commercially available tools opting for a Tcl [2]-based scripting approach. From a today's perspective, Tcl does not always provide the best user experience. In addition, the implemented techniques are usually vendor-centric and do not work well in a diverse development environment.

For integration purposes with other existing tools -- which might use other scripting languages -- a more language-agnostic approach is desirable in order to provide maximum flexibility. This is driven by the experience that the preferred language depends on the user, problem, and situation [3].

The here proposed abstraction enables SystemC simulation engines to interface with a wide variety of scripting languages and allows extending their capabilities by third-party utility APIs. The presented solution is therefore as flexible as possible: due to reusable integration code, the integration of other languages is enabled while requiring minimal code-rewrite efforts.

Based on the proposed methods and techniques, we were able to quickly build Python and Tcl interfaces to our API abstraction. This was tested with Accellera's SystemC simulator [4] and QuestaSim [5].

The remainder of this paper is structured as follows: In Section 2, related scripting solutions are compared. Section 3 describes our abstraction API, enabling simple wrapping of third-party base-class APIs in a scripting-language-agnostic way. This section is subdivided into four subsections: (3.1) a description of Simulation API, (3.2) the design of the language-independent interface including delegation and plugin API, (3.3) a wrapper of the AMBA Plug-and-Play API into a base class, and finally (3.4) a wrapper for Cadence `scireg` [6]. It is complemented by the presentation of three language implementations as explored in Section 4, (4.1) Python and (4.2) Tcl, followed by a Python implementation in QuestaSim, with conclusions being drawn in Section 5.

2. Related Work

The integration of scripting languages for control purposes into EDA tools is quite common, particularly in the case of Tcl [7]. It is simple to learn and a good choice for sequential scripting of work procedures.

Even in academia and open source tools, scripting languages are introduced frequently [1], although the manner of integration and the intention of usage are quite different. There are several reasons for scripting integration: for (a) automating procedures, (b) configuring simulation parameters [8], (c) assembling the simulation [9], (d) extending the simulation with high-level models [10], (e) easy test integration, and (f) interactive introspection. The most common use-case for a scripting language is a), i.e. easy automation of procedures like building, testing, or synthesis. This is employed in a vast variety of tools.

For configuration purposes, there e.g. a Lua [11] integration in GreenLib [12] exists. It handles parameter loading for their *CCI* [13] implementation *greensocs-greencontrol*. The platform designer can insert the loading of different Lua scripts at certain points of the simulation in order to change the *CCI* parameter configuration of the instantiated models.

To assemble a simulation platform from a component library, most commonly a top-level model or direct instantiation in `sc_main` is used; it always distills to a list of components, parameters, and connections. This is suited perfectly for loading from external configurations such as e.g. IP-XACT from Kaktus2 [14]. Tools like *Synopsys Platform Architect* use a GUI-generated XML file. ReSP [9] uses a complex model-parsing mechanism with help of the *gcc-xml* compiler [15] and Python wrapping of C++ components for assembling a platform in Python. Due to its requirement of a fully integrated build-flow it is therefore not applicable to other tool-chains.

The integration of real RTL hardware into a SystemC/TLM simulation can be achieved by integrating an RTL IP core into a TLM simulation wrapped in transactors handling the abstraction-level shift. Towards high-level models, several ways of components integration exist: The most common way is synthesizing a SystemC/TLM model from a functional model, for example from Matlab [16]. A more direct integration with more integration possibilities and even more fine-grained selection of timing and functional granularity is possible with GreenScript [10]. It is designed for integrating TLM models written in Python, enabling fast development without the need to recompile as well as testing in an accurately modeled system.

Stimuli and test code does not need to be synthesizable. Instead, this code is targeted to be reused as often as possible on different abstraction levels. Several languages are targeting this gap like *e* and *SystemVerilog*, which are typically used by commercial vendors to specify test patterns for models of different abstraction levels. The shortcomings of these solutions are usually on the upper end, as they do not support functional modeling languages like Matlab. In contrast, an abstraction level as offered by Matlab, in term, might turn out problematic when used for describing strict timing and parallelism. Using Python for generating test patterns can overcome these shortcomings [17].

While developing and debugging models, the need for inspecting the inner workings in order to understand the procedures is important. Some tool vendors have support for interactive introspection, but most of them use debuggers like *gdb* or introspection via *CCI* accompanied by proprietary methods. Due to the complex usage of these features and the vendor-specific design, it comes down to tracing and *printf*-debugging most of the time.

Resulting from our work on the SystemC-based full-system simulator *SoCRocket* [18, 19], the tool presented in this paper was originally developed for versatile automation procedures, configuration, and assembling of platforms in diverse environments.

Within *SoCRocket*, a multitude of tool environments [4,5] and libraries [6,12,20,21] were used. In the course of the work, others eventually replaced certain tools and libraries, leading to the situation that now a large number of SystemC/C++ libraries from various vendors and with varying programming styles reside within *SoCRocket*. On top of that, additional tools were designed, such as *SignalKit* providing high-level signal routing, *VerbosityKit* providing a logging system, and AMBAPnP providing *Grlib*'s AMBA Plug and Play functionality. For enabling dynamic use of all available introspection tools and APIs we introduced a scripting language and appropriate interfaces into our infrastructure. With the diverse nature of *SoCRocket* in mind, a primary design goal was that it would not interfere with our model libraries in order to maintain interoperability with different simulators.

For scripting, we chose Python [22] for a number of reasons: firstly, Python itself is a very dynamic language, suitable for simple scripting automation up to complex object-oriented or functional programming. Moreover, we use *waf* [23] as our build system, which was chosen together with ESA at the beginning of the *SoCRocket* development; hence, all our build scripts were written in this language. In addition, we use Python for a lot of

other tools and scripts as well, thus Python was a natural choice as scripting language for *SoCRocket* in combination with the used SystemC simulators (Accellera OSCI [4], Mentor QuestaSim [5]).

Therefore it is important to integrate a scripting language into the existing SystemC code, which does not require any changes to the models and utilizes as much as possible the various APIs of the base classes. Base classes are the building blocks, which allow easy code reuse in SystemC simulations. We identified following classes to be used frequently in our SystemC designs, which offer an API worth to be accessible from a scripting interface: `sc_object`, `scireg`, `AHBDevice`, `APBDevice` and `gs_config`. Due to our flexible approach, interfaces to other classes can be created as well, without big efforts.

3. USI API Abstraction

The proposed universal scripting interface (USI) consists of three parts: firstly, a *simulation API* to be integrated in the platform execution code. Its purpose is linking in the platform execution so that users may hook into different simulation phases (for example `start_of_simulation`). Secondly, in the *Language Independent Interface*, the delegation object aggregates the interface implementations and creates usable SWIG-object proxies for the scripting environment. This enables language independent tool implementations. A plug-in API, allowing third party utilities to implement functions on top of every `sc_object` base class, completes the set.

3.1 Simulation API

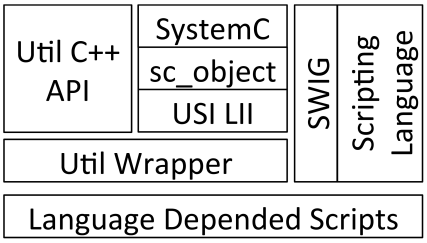

Figure 1: USI builds an abstraction on top of SystemC and SWIG for the generation of language-independent scripting wrapper

The simulation API is meant to be used during the startup of the platform with the functions needing to be called in the `sc_main` function. Before a simulation model is created, the commands in Listing 2, lines 1 to 4, need to be executed. The first line ensures that the needed utility is linked and loaded with the simulation. The second initializes the scripting core and creates an `sc_module` to hook into the SystemC phases and prepares the whole scripting environment. The `usi_load` function can be called anywhere in SystemC, it loads and executes a script. Scripts can either be called with the full path or as a language-internal package, if it is available to the simulation environment. The fourth function calls the first internal USI phase and executes the code of all utilities that need to execute something prior to creating the simulation models. This enables utilities to prepare internal data structures.

This is followed by model instantiation and simulation-run preparation. The simulation run is done by a special call to `usi_start,` as seen in Listing 2, line 5 to 7. This function invokes the code in Listing 1. It can be used analogous to the SystemC function `sc_start`. In addition to pause control, the function is registering a system signal handler and calling the `end_of_initialization` phase. Moreover, after the simulation has stopped, two evaluation phases are triggered to execute post-processing scripts, i.e. process retrieved simulation data and clean up.

This API allows hooking scripts into any simulation phase as described in the previous chapter. The interaction with the simulation is done via third party utilities, which in the simplest case can be building directly upon SWIG.

Directly built upon SWIG is our adaptation of the *GreenControl* API. It enables reading from and writing into various model parameters during simulation. Furthermore, it allows to register callback functions on read and write operations of these parameters. It is a reimplementation of the introspection aspects from the *GreenControl* parameters in *GreenScript*. The *GreenScript* implementation is only usable with Python and therefore not easily portable to other languages.

```
1  sc_status status = SC_RUNNING;
2  while (1) {
3    if (status == SC_RUNNING) {
4      sc_start();
5    } else if (status == SC_PAUSED) {
6      usi_pause_of_simulation();
7    } else {
8      break;
9    }
10   status = sc_get_status();
11 }
```

Listing 1: USI `sc_start` replacement

To overcome language dependencies, the Language Independent Interface (LII) is introduced. The LII is the core of the USI implementation and encapsulates all language-specific details, providing an abstract foundation for third party utilities like GreenControl. To accomplish this, the LII builds upon SWIG and implements service functions in C++ and in the desired target language as shown in Figure 1.

```
1 USI_HAS_MODULE(cci);
2 usi_init(argc, argv);
3 usi_load("usi.api.cci");
4 usi_start_of_initialization();
5 usi_start();
6 usi_start(sc_time);
7 usi_start(double, sc_time_unit);
```

Listing 2: Simulation API, functions to execute in `sc_main`

3.2 Language Independent Interface

The main aspect of LII is providing a unified way to access simulation objects from the scripting environment via its hierarchical path. The selected object is wrapped into a `USIBase` object, which has almost no features of its own, but allows dynamically casting the linked `sc_object` to any interface class registered from any utility. These interface classes get translated by SWIG into a proxy object in the scripting language.

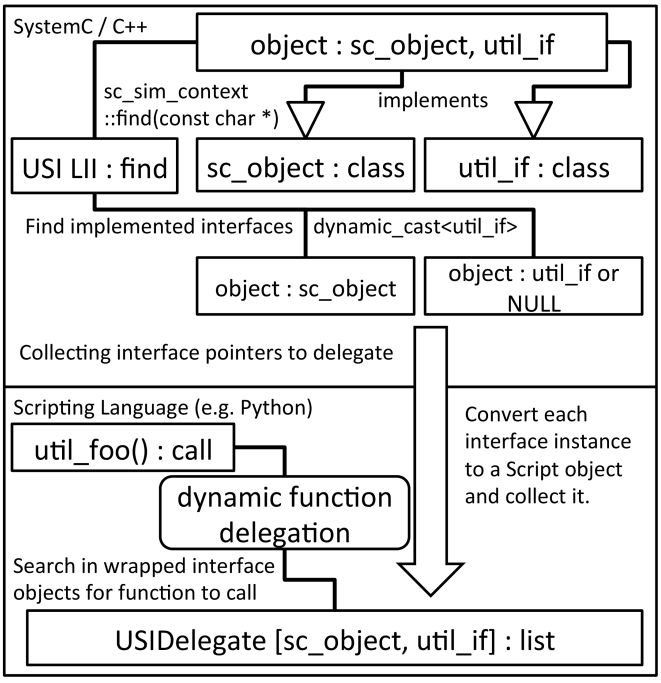

Figure 2: Dynamic delegation of the target language and dynamic casts make the interface language-agnostic

Therefore, third party utilities can extend the scripting API as plug-ins. To do so, each plug-in registers a generator function to return an object of a SWIG-enabled interface class for a certain hierarchical path or `sc_object`. These objects will use SWIG for converting the dynamic pointer to the target language and collect the different proxy classes referring to the interface instances of the initial `sc_object`.

In the target language, a delegation object is created that corresponds to the `sc_object` or hierarchical path. It is used as wrapper for the list of interface instances and gives unified access to the methods of the collected APIs.

This architecture is designed with the interface pattern in mind: a class can be inherited from any number of different interfaces to implement the corresponding functions. This allows others to cast the object to an implemented interface and use the interface as designed by the interface developer, regardless of how or what the object is or does.

In the simplest case an `sc_object` is dynamically cast to an interface like, e.g., `sc_module` or SoCRockets AMBA Plug-and-Play Interfaces. If this cast returns `NULL`, the interface is not implemented on the `sc_object`; otherwise it is implemented and will create a SWIG proxy object. An object can implement multiple of such interfaces. Within the delegation process, all the proxy objects for such interfaces will be collected. To ease this process, a C++ macro can cover this whole process, but only in case it is a real interface of the class as illustrated by Listing 3, line 4. Otherwise, a more complex approach must be taken.

```
1 USI_OBJECT
2 USI_INIT_MODULES()
3 USI_REGISTER_OBJECT_GENERATOR(funct)
4 USI_REGISTER_OBJECT(type)
```

Listing 3: Plugin API

Some utility APIs can only interact with their counterparts in SystemC models through the hierarchical path name or the `sc_object` pointer. The information could be in a data store not reachable by the hierarchical path. To make such utilities work, we provide the ability to define a generator function (see Listing 3, line 3). It can cope with any kind of function, creating an interface object from either an `sc_object` or a hierarchical path. To demonstrate the API, the AMBA Plug-and-Play interface and the `scireg` base class integrations are explained below.

4.3 AMBA Interfaces

```
1  %module amba
2  %include "usi.i"
3
4  USI_REGISTER_MODULE(amba)
5  %{
6  #include "amba.h"
7  #include "ahbdevice.h"
8  #include "apbdevice.h"
9  %}
10 %include "amba.h"
11 %include "ahbdevice.h"
12 %include "apbdevice.h"
13 %{
14 USI_REGISTER_OBJECT(AHBDevice);
15 USI_REGISTER_OBJECT(APBDevice);
16 %}
```

Listing 4: AMBA USI/SWIG interface file

The information stored in the Gaisler AMBA Plug and Play module is available through an abstract interface base class for AHB and APB Devices. Due to the USI Plugin API, only the headers of the base classes need to be included; a USI Module needs to be declared and the interface classes need to be registered. This is done with the code in Listing 4. `USI_REGISTER_MODULE` is creating a function returning either a SWIG proxy object of a corresponding AMBA device interface of a specific `sc_object` or `NULL`. An `sc_module` can implement any

of an AHB Device or an APB Device, and in the scripting language the corresponding functions are available for the delegation object with the same hierarchical path as the `sc_object`.

3.4 scireg

In order to be able to integrate the Cadence `scireg` API, more effort is needed: `scireg` not only expects a publisher/subscriber API and, furthermore, is not based on the hierarchical paths in general. This API enables tools to register for subscription before simulation models get created, so they get notified when a new `scireg` class is instantiated. The USI plugin for `scireg` has to be implemented as a registry to such notifications; hence, we have to build a database of `scireg` devices. Every `scireg` device stores a link to its parent `sc_object`. This parent object is used as key for the USI abstraction. We implemented the helper class `scireg_parent`, containing a function returning all `scireg` instances corresponding to a certain parent. We use a lookup function to return the SWIG proxy object or `NULL` if no parent is found. It is registered using `USI_REGISTER_OBJECT_GENERATOR` as shown in Listing 5.

```
1  USIObject find_scireg(sc_object *obj, string name) {
2    scireg_parent *instance = registry::find(obj);
3    return (instance) ?
4      SWIG_NewPointerObj(
5        SWIG_as_voidptr(instance),
6        SWIGTYPE_p_scireg_parent,
7        0)" :
8      NULL;
9  }
10 USI REGISTER OBJECT GENERATOR(find scireg);
```

Listing 5: scireg USI/SWIG interface function

4. EXAMPLE IMPLEMENTATIONS

We verified our approach by testing the designed API in Python, Ruby, and Tcl. Furthermore, we used the Python API in conjunction with the QuestaSim simulator.

4.1 Python

Due to the fact that all our supporting scripts and the build-system are written in Python, Python was the logical choice to use for the direct scripting integration and, as an implicit result, our Python implementation is by far the most complete.

Object delegation comes natural to Python. The implementation can be seen in Listing 6: the method `__getattr__` implements delegation, and the function `get_if_tuple` returns a tuple of valid SWIG proxy objects of C++ interfaces implemented on a corresponding `sc_object`. This method can be implemented directly in SWIG C++ for unifying the `InterfaceDelegate` with the SWIG proxy object. The `__dir__` function is needed to enable command completion within the interactive shell.

On top of this API, our Python implementation grew into a library with over 2000 lines of code (LOC) in 32 files, building a reusable scripting infrastructure with features including but not limited to simple and extensible argument parsing, platform configuration from JSON, register as well as memory access, and power measurement.

In the following paragraphs, we describe a few examples of complexity reduction with Python. The first example is a tool for power measurements. The original C++ source code has 257 LOC; after we re-implemented it in Python it was reduced to 65 LOC. We maintained equal usability and functionality. Moreover, the Python implementation is more flexible in terms of extensibility and output formats. Aside from direct text output (`stdout`), exporting parameters to files in several formats (JSON, CSV, XLS) is possible as well. Furthermore, it is possible to dynamically load parameters at runtime and reconsider earlier configuration decisions.

A second example is the loading and storing of platform configurations. The C++ implementation has more than 800 LOC. It only implements loading and storing the configuration in JSON. The Python implementation builds the exact same behavior in 16 LOC due to extensive use of the *Python standard library*. The simple task of loading or dumping a platform configuration can be accomplished in one line in the interactive Python shell.

Moreover, the used format can be chosen freely: the Python standard library has support for *JSON*, *XML*, *CSV*, and *INI*. Bindings for several databases (*MySQL*, *MongoDB*, *HDF5*) are available as well.

```
def __dir__(self):
    result = set()
    for iface in self.get_if_tuple():
        result.update(dir(iface))
    return sorted(result)

def __getattr__(self, name):
    result = None
    for iface in self.get_if_tuple():
        result = getattr(iface, name, None)
        if result: return result
    super(InterfaceDelegate,
        self).__getattr__(name)
```

Listing 6: Python implementation of the USI object delegation

Other simple one-liners are, for example, hex dump generation of any memory area or register set, writing a file to memory, registering callbacks on register (*scireg*) or parameter (*GreenControl*) changes, or dumping the Grlib AMBA plug-and-play configuration.

All example tasks use heavy string handling to work with the hierarchical SystemC names. They are quite complicated and error-prone in C++, but come naturally in most scripting languages. Moreover, the tasks are usually performed before or after simulation execution and therefore do not impact simulation speed. An exception is the use of callback functions, so these should be used with caution.

4.3 Tcl

The *Tcl* implementation is only intended to validate the USI delegation. Unlike in Python, in Tcl object-oriented programming is not a natural feature of the language. Much like the object orientation in the popular *GTK+* C library, it is just a coding style build on top of libraries utilizing nested name spacing. One implementation is *[incr Tcl]*, which is part of the language implementation by now. SWIG uses the *[incr Tcl]* style to wrap C++ classes to *Tcl* so it is fully compatible with this coding style. Each class and object by itself is a namespace, and each namespace can implement a method `unknown`, which is invoked whenever a member of the namespace, e.g. an object, is not found. The Tcl interpreter is easy to integrate and extend through a very well documented API, which definitely is the result or resulted in the heavy use by EDA tools.

4.3 Python in QuestaSim

To verify interoperability, we tested whether the proposed tool APIs could also be used in QuestaSim. For the following test, Python was chosen to integrate with QuestaSim, but Ruby could have been integrated as well. To make use of Python in QuestaSim, quite a few restrictions need to be overcome: firstly, QuestaSim comes with its own SystemC compiler, which is at the time of writing a 32-bit *gcc* in version 4.7.4; we therefore require a 32-bit version of the Python library to link against. Secondly, QuestaSim does not support the use of `sc_main`. They need to have all SystemC models exported by an `SC_MODEL_EXPORT` macro, which basically creates a magic function beginning with `mti__` followed by the class name and creates a new instance of the class with error-handling support. We introduce the macro `USI_MODEL_EXPORT` calling our USI API function, but we restrict it to be only called once in a simulation. As a caveat of this solution, it must be remembered that this approach uses an internal Questa API that may change at any time. Moreover, we cannot start the simulation on our own as the simulation control is integrated into the QuestaSim core. Instead of calling `sc_start`, we have to call `run` in the Tcl shell of QuestaSim. We cannot wrap that command, as an interactive Python shell is not possible, but we can issue Python commands by wrapping them into PLI functions that can be called by the Questa Tcl shell. Callbacks have to be registered with `sc_dpi_register_cpp_function`. QuestaSim ships with its own Tcl interpreter. A more direct integration with it would be desirable and has to be investigated further.

5. CONCLUSION

We have shown that our proposed API greatly eases the integration of third party C++ APIs. Furthermore, it enables the use of different scripting environments in SystemC simulation engines for a vast amount of possible applications, including manipulation, analysis, and setup. It unifies access to the base classes and achieves fast development. Moreover, vendors can provide their APIs in a language-agnostic way, leaving the choice of the scripting language to the user. This enables developers to easily do complex instrumentation and analysis of simulation runs. Our solution results in a drastic reduction of complexity in terms of lines of code, for example reading and writing JSON configurations could be reduced from over 800 lines of C++ code to fewer than 16 lines of scripting code. The source code of our solution is published under LGPL [26] at [27].

In future work, we will further extend the interface by a better QuestaSim integration as well as support for the Cadence Virtual System Platform. We are actively looking into the integration of additional third-party APIs, such as the TrapGen [25] operating system emulation in order to register scripting functions as processor intrinsic and sr_report [28] integration as an additional data source. The USI abstraction builds a solid foundation: it speeds up the development process of SystemC simulation environments by enabling the developer to facilitate the usage of various features of numerous scripting languages.

REFERENCES

[1] P. Chen et al., “Scripting for EDA tools: a case study,” in Quality Electronic Design, 2001 Int. Symposium on, 2001, pp. 87–93.

[2] B. Wheeler, Tcl/Tk 8.5 Programming Cookbook. Packt Publishing Ltd, 2011.

[3] A. Van Deursen et al., “Domain-specific languages: An annotated bibliography.” Sigplan Notices, vol. 35, no. 6, pp. 26–36, 2000.

[4] I.-S. S. Board, “IEEE Standard for Standard SystemC Language Reference Manual,” IEEE Std 1666-2011, 09 2011.

[5] Questa SIM User’s Manual, Mentor Graphics Corporation.

[6] S. Swan and J. Cornet, “Beyond TLM 2.0: New Virtual Platform Standards Proposals from ST and Cadence.”

[7] E. Todorovich and O. Cadenas, “TCL/TK for EDA Tools,” in Programmable Logic, 2007. SPL ’07. 2007 3rd Southern Conference on, Feb 2007, pp. 107–112.

[8] C. Schröder et al., “Configuration and control of SystemC models using TLM middleware,” in Proceedings of the 7th IEEE/ACM International Conference on HW/SW Co-design and System Synthesis (CODES+ISSS), 2009, pp. 81–88.

[9] G. Beltrame et al., “ReSP: A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration,” Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 28, no. 12, pp. 1857–1869, Dec 2009.

[10] J. A. M. Berrada, “SystemPython: a Python extension to control SystemC SoC simulations,” GreenSocs meeting presentation, Design and Test in Europe Conference (DATE), 04 2007.

[11] R. Ierusalimschy et al., “Lua 5.2 reference manual,” 2011.

[12] R. Günzel, GreenSocket – Conceptual Details, GreenSocs, 04 2010.

[13] Accellera, “Accellera working group for Configuration, Control and Inspection,” Website http://www.accellera.org/activities/committees/systemc-cci/, 2015.

[14] A. Kamppi et al., “Kactus2: Extended ip-xact metadata based embedded system design environment,” in Embedded Systems Week/MeCoES: Metamodeling and Code Generation for Embedded Systems workshop, Tampere, Finland, 2012, pp. 17–22.

[15] B. King, “GCC-XML,” Website: http://www.gccxml.org/HTML/Index.html, 2015.

[16] W. Hassairi et al., “Matlab/SystemC for the New Co-Simulation Environment by JPEG Algorithm,” MATLAB–A Fundamental Tool for Scientific Computing and Engineering Applications, vol. 2, pp. 120–138, 2012.

[17] J. Villar et al., “Python as a hardware description language: A case study,” in Programmable Logic (SPL), 2011 VII Southern Conference on. IEEE, 2011, pp. 117–122.

[18] L. Fossati et al., “Socrocket: A virtual platform for soc design,” 2013.

[19] T. Schuster et al., “SoCRocket - A virtual platform for the European Space Agency’s SoC development,” in Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on, May 2014, pp. 1–7.

[20] J. Gaisler et al., “GRLIB IP core user’s manual,” Gaisler research, 2007.

[21] K. Cheung, “TLM-2.0 Kit for AMBA,” Website http://edablog.com/2011/02/17/virtual-mode-reuse/, 02 2011.

[22] Python Language Reference, Python Software Foundation, 02 2015.

[23] T. Nagy, The Waf Book, 2014.

[24] S. Wrapper and I. Generator, “Swig,” Website http://www.swig.org, 2014.

[25] L. Fossati, “IP-SOC 2010.”

[26] R. M. Stallman, "GNU lesser general public license," *GNU Project–Free Software Foundation, Website: http://www.gnu.org/licenses/lgpl.html* , 07 2015.

[27] R. Meyer, “Universal Scripting Interface for SystemC: Python Implementation,” SoCRocket Group, TU Braunschweig, Website: https://socrocket.gitub.io/pysc/ , 07 2015.

[28] J Wagner, R. Meyer, R. Buchty et al., “A Scriptable, Standards-Compliant Reporting and Logging Extension for SystemC,” 3rd Workshop on Virtual Prototyping of Parallel and Embedded Systems (ViPES), 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV), 07 2015.