Embed Scripting inside SystemC

Joël Vennin, Stéphane Penain
Prosilog
8, rue traversiere
95000 Cergy, France
vennin@prosilog.com, penain@prosilog.com

Luc Charest, Samy Meftali and Jean-Luc Dekeyser
LIFL - USTL
Cité Scientifique, 59655 Villeneuve d'Ascq cedex, France
meftali@lifl.fr, charest@lifl.fr and dekeyser@lifl.fr

Abstract

Embedded system designs and simulations become tedious and time consuming due to the complexity of modern applications. Thus, languages allowing high level description, such as SystemC, are more and more used. We present in this paper a new methodology allowing scripting inside SystemC. We integrate both SystemC and Python within a single framework for system designs and simulations called SystemPy. Communication is performed using a Simple Wrapper and an Interface Generator(SWIG). SystemPy allows dynamic IP changes during the simulation. This makes designers able to perform a quick architecture exploration without stopping the simulation process. Steps and performances of our framework are illustrated on mixed SystemC - Python system.

1 Introduction

Software and hardware architecture research aim generally at reducing costs of the platform creation process and decreasing time to market. Respecting these constraints, several works have been realized around existing problems in software and hardware domains.

We define as dynamic interaction the possibility we give to the end user to perform momentary simulation interruption, component replacement or new test bench insertion without even having to stop the simulation process.

Nowadays a set of existing ADLs (Architecture Description Language) allows system design at different abstraction levels from the functional to the register transfer (RTL) one. They are especially made for a specific domain while classical languages are more generic. Unfortunately, most of these languages do not provide the possibility to interact with them dynamically in an easy way. Thus, this lack of language flexibility makes the interconnection of several components together also known as architectural phase, very painful and error prone.

We find for instance this kind of problems in SystemC [Ini, Amo00]. SystemC is a hardware description language, appeared in 1999 as a new language for hardware description in system design. It is a C++ library which allows hardware, software and system-level modeling. SystemC simulators become more and more mature including new functionalities. In fact, the first version was RTL system design oriented. With SystemC version 2.0, abstract layers have been added to SystemC in order to get more and more design abstraction [Pan01]. These enhancements brought transaction level modeling capability to the initial language specification.

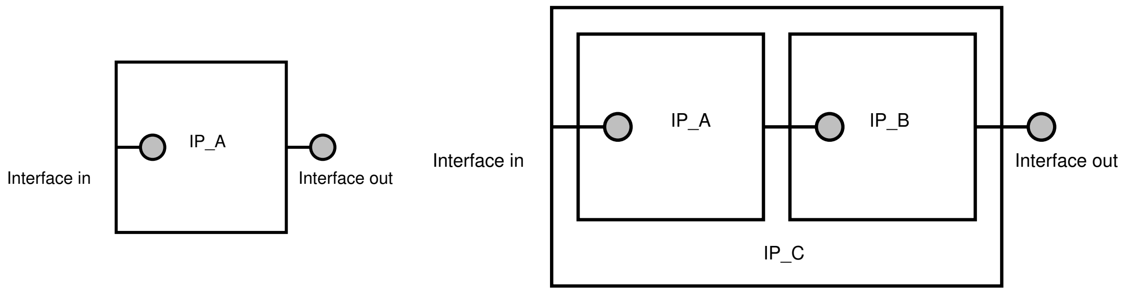
SystemC however falls into the category of languages where the compositional step is tedious. Of course, it is possible to use a framework to realize the compositional step, but because of the underlying use of C++, the end user is forced to go through a compilation step taking lots of time and to handle complex Makefiles.

Another negative point, in SystemC is the difficulty to interact dynamically during the simulation step. Furthermore, even if SystemC allows designer to describe a system at TLM (Transactional Level Model) or RTL level, it does not provide the basic features to realize multi-level simulations.

One of the improvements that we could imagine for system level design languages in general and for SystemC in particular is to mix it up with a scripting language in a single design framework taking in charge languages interactions.

Scripting, using Perl, Python or TCL for instance, inside a system level description language will most likely allow designers to benefit from the easy way of handling and writing algorithms, the high productivity of scripting languages, and consequently to reduce dramatically the design time to market. In fact, the developing time of an application using scripts is four times shorter than using C++ for example [Pre00, Ous98]. In this paper, we show that the integration of a scripting language inside SystemC is a necessity to speed up design and simulation process. We propose also a way to implement it dynamically using Python.

This paper is organized as follow, the Section 2 reviews the related works: The conception and the realization of SystemPy are described in Section 3. Then, a validation of our methodology is illustrated by an example in Section 4. Finally, we conclude this article in Section 5.



(a) *IP_A can be connected to the global system via its interfaces.* (b) *Hierachical composition of components, IP_C is a composition of IP_A and IP_B.*

Figure 1: *View of IP as component, interfaces are represented in a UML way, by using circles.*

2 Related Works

This work is based around component technology and script programming inside SystemC. First and foremost there is a definition for a component: *"a component is a non trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces."*[Kru98]. Actually, an IP (Intellectual Propriety) may have a complex system independent implementation and provides interfaces to plug it in the system (e.g. Figure 1(a) and Figure 1(b)). The functionality of a component can be realized by assembling small components. So the creation of new IPs is based on the composition of other IPs. This can be achieved dynamically or statically. A key advantage of the dynamic composition over the static composition, is the possibility to change the component composition at runtime. Thus, one should be able to change an IP with a different one during the simulation (e.g. plug/unplug USB device).

Different approaches exist to compose a component or to realize a system. One of the easiest way is to use a graphical interface allowing the interconnections of blocks such as

VCC [BFSVT00]. Nevertheless large and complex systems are difficult to manage, and most of the compositions are static. Static means that the composition is defined and hard-coded in the source files. Another approach is to use the programming based composition, in this approach the current ADL is used to realize the composition. This approach is a tedious task and it is done statically in most ADLs.

The use of scripts offers a very good trade-off flexibility performance for nearly all the applications domains such as EDA (Electronic Design and Automation) which is our main focus in this article (e.g. tcl scripts are widely used by Synopsys). In Balboa [DSGO02], script is the base of this environment. It proposes methods to ease the composition step of a design allowing introspection and reflection [DSG03]. Nevertheless, Balboa does not offer the possibilities to script new component to reduce development time but it supports type checking mechanism that offers possible automatic connections between IPs. So, to reach our objectives we have decided to use SWIG [Ous98]. Such a tool is able to parse C and C++ code and give us the interfaces for a given language. Despite the points developed in [DSGO02], we will demonstrate in this article that SWIG is capable to navigate into our system and class hierarchy in an easy way. Our approach is different from the Balboa one. We prefer to make the realization of benchmark or the conception of algorithms easier in order to raise the system abstraction level.

The following sections present the SystemPy framework that is based on the SystemC simulator powered by scripting capabilities.

3 The SystemPy Simulator

The SystemPy framework is a combination of the SystemC description language and the Python scripting language. This framework provides a set of functionalities to make the use of SystemC easier. The following sections describe the overview of the framework, then we will present the required modifications of the SystemC kernel to support scripting.

3.1 Framework Overview

The framework is based on several existing tools and languages. Thus, we have defined a methodology and created a set of functionalities to improve the integration of the existing tools and languages. The methodology is split into three steps: the IPs creation step, the system composition and the simulation step. Each of the steps are described in details here after.

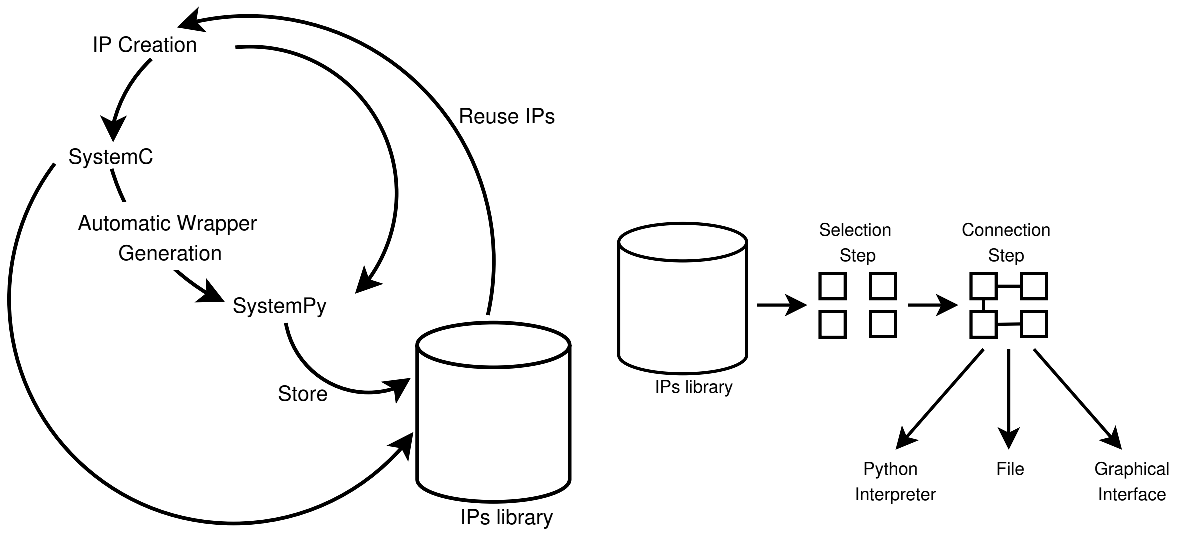
IPs Creation Step: Figure 2(a) shows all parts of the creation step. When one decides to build a system he can reuse an IP from the available libraries. Unfortunately, sometimes it is necessary to create a new IP. Depending on the designer's needs, the creation step offers a choice to create a custom IP in SystemC language or in Python language. Actually, if one needs to create an efficient IP in term of simulation time, he has to create it using SystemC, otherwise he can use the SystemC Python binding. It is well known that scripts are slower than compiled languages even if we can reach good enough performances with a JIT (Just In Time Compiler) like psyco [Rig04]. A designer using scripts to create his system can reuse IPs from SystemC or Python libraries whereas a designer using compiled IPs is limited to SystemC ones. Moreover, when he uses the Python script, he can also inherit SystemC IPs inside scripts. This possibilities are showed in Section 4.1.

The use of SystemC IPs combined with Python IPs is possible through the use of SWIG interfaces. Actually, when one creates a SystemC IP, SWIG is called in order to generate a proxy between the C++ and the Python language (commonly called a wrapper).

The use of C++ implies a compilation step to compile the SystemC module and its wrapper. We are using the SCons build mechanism [Kni] instead of the complex Makefile mechanism, so the generation and compilation steps are automatically done by SCons. Moreover, SCons is written in Python, so it is directly integrated inside our framework.

The reuse capabilities offered by the framework is possible via the library manager. In fact, all available SystemC and Python IPs are stored inside this library. All elements or IPs in this library can be reused during the creation step or during the System Composition step.

System Composition Step: The System composition step is similar to the creation step. The difference is that inside this step one defines the *toplevel* of your system. The *toplevel* is realized using the IP library. So, during the composition step, one can select IP written in SystemC or in Python. Once the selection is done, one needs to interconnect all IPs together. As we can see on the Figure 2(b), one can realize his system using the Python interpreter, then save it in a file or even view it using a graphical editor. Although planned, at the time of the writing this article, the graphical editor is not available yet.



(a) *The different steps of the IP creation stage, the designer can reuse the IPs from a library or write them directly in SystemC or Python.*

(b) *The system composition step, IPs are selected from a library to be saved, displayed or sent to the Python interpreter.*

Figure 2: *The first two stages of our framework.*

Simulation Step: The simulation step consists of five states along with their associate simulation control commands and our new dynamic IP control commands (see Figure 3). All commands can be executed using the Python interpreter. When we start the framework, we are in the *Initial* step which means that there are no loaded modules. To go to the *End of Instantiation* state the designer has to import the designed system. Next, the designer is able to start the simulation. During the simulation, he can interrupt it to pass in the *Edit* state which allows him many operations. The first possible operation is the capability to add or remove signals from the SystemC trace manager. The second available functionality in this state is to add or remove connections between modules. The third command provides add and remove module actions. So, it is possible to add, remove or replace an IP after the simulation start. The last functionality allows the change of internal values within all objects of the designed system.

Thus, this new functionalities offers the control of virtually anything inside the SystemC simulator. We show in the Section 3.2 a subset of changes inside the SystemC kernel to embed the script capability and support the new simulation commands.

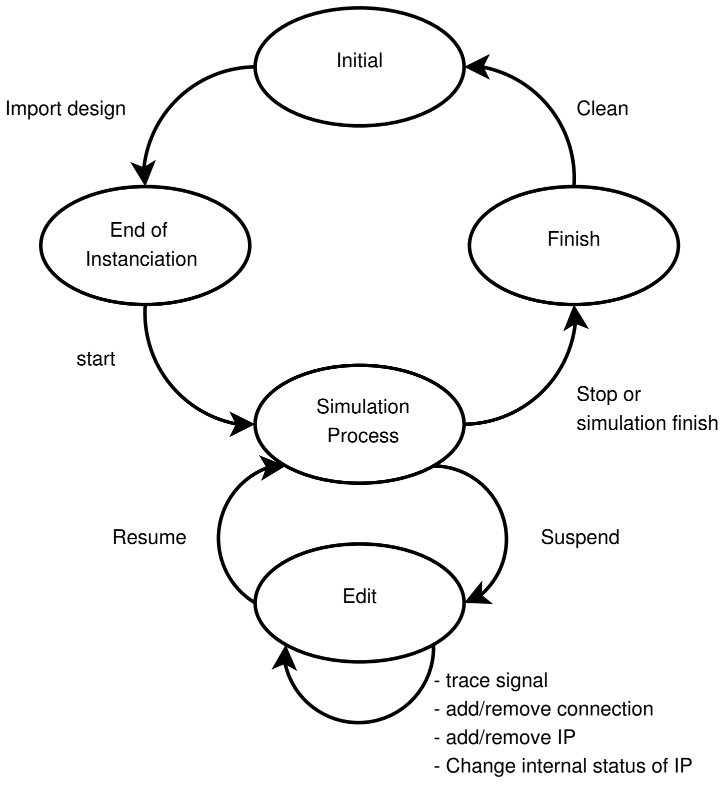


Figure 3: *All possible state of the simulation step, our solution introduce the possibility to modify the model on the fly.*

3.2 SystemC Kernel Modification

Our framework is based on SystemC and Python languages, a lots of changes have been required to allow interoperability between these two languages. In the following sections we highlight a set of changes inside the SystemC kernel.

3.2.1 Wrapping SystemC

Wrapping a language to another one can be done manually, but existing tools can generate it automatically. SWIG allows us to generate wrappers from SystemC to a scripting language. However, in some cases, we need to add specific information to the wrapper generation. In fact, using a SWIG configuration file for each class allows us to decide which one among all of the class methods should be exported to the script API. Moreover, in a SWIG configuration file, we can add C++ code to create helper methods or functions that enrich the script API with high level functions.

The Listing 1 explicitly defines a set of accessible functions in the script. Thus, it is possible to get the simulation time by executing the `sc_simulation_time` function in the Python interpreter.

Wrapping is not only the call of functions in an interpreter, it is also a way to inherit C++ class in Python. In our opinion, this feature is really interesting. Indeed, it allows us to create new SystemC flexible modules in Python. In the following section we present how to wrap the `sc_module` classes.

```
%{
  #include "systemc.h"
%}

%include "sc_time.i"

void sc_start( const sc_time& duration );
void sc_start(double duration);
void sc_stop ();
double sc_simulation_time();
...
```

Listing 1: *SWIG configuration file for the `sc_simcontext.h` SystemC header file.*

3.2.2 Special Case: `sc_module`

The `sc_module` class contains a set of virtual protected methods. It is necessary to allow a Python class to override these virtual methods. However, it is not possible to have access to C++ protected attributes and methods from the script side. To resolve this issue, we have created a class inherited from `sc_module` that allows access to protected members. It is illustrated in Listing 2.

Since SWIG version 1.3.20, the overriding is supported by using the `director` key word. It allows to override C++ methods by scripting methods. The `director` key word corresponds to a feature in the SWIG configuration file, which is disabled by default. Hence we need to enable it. The Listing 3 shows how to define the SWIG configuration file for the newly created `sc_module_swig` class.

Some other changes around the `sc_module` class have been done. The `SC_MODULE` macro has been modified to register each module as a SWIG module. It has been done to make the handling of modules easier inside the script language. The following section describes a part of the complex work realized to support the `SC_METHOD` and `SC_THREAD` macros.

3.2.3 Kernel Modification

SystemC provides concurrent behavior using `SC_METHOD` and `SC_THREAD` macros. They must be made available in Python to allow the creation of SystemC process. We show in the Figure 4 a light class diagram of SystemC process classes.

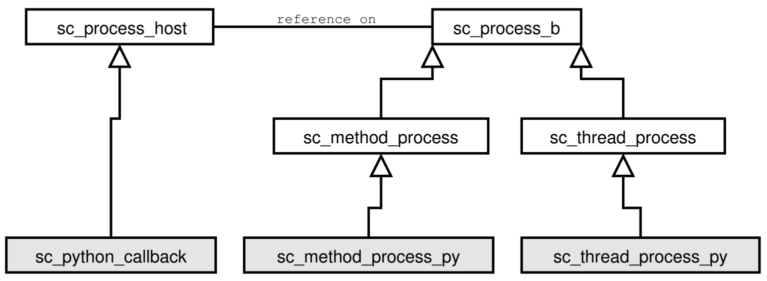


Figure 4: *Process class diagrams with our added classes shown with gray background.*

The `sc_process_b` is the base class for the process manipulation in SystemC. It contains a set of attributes to handle process mechanism. Moreover, it contains the `execute` method that call the `run` method of the associated `sc_process_host` object. The default implementation of the `sc_process_host::run` method is to call the method passed by parameter using the

```
class sc_module_swig : public sc_module {
 public:
   sc_module_swig (const char * nm)
     : sc_module ((sc_module_name)nm){}

   virtual void beforeEndOfElaboration(){}
   virtual void endOfElaboration(){}
   virtual void startOfSimulation(){}
   virtual void endOfSimulation (){}

   inline void dontInitialize ()
     {sc_module::dont_initialize ();}

   inline sc_sensitive & getSensitive ()
     {return sensitive;}
   ...
 protected:

   inline void before_end_of_elaboration()
     {this->beforeEndOfElaboration ();}
   inline void end_of_elaboration()
     {this->endOfElaboration ();}
   inline void start_of_simulation()
     {this->startOfSimulation ();}
   inline void end_of_simulation()
     {this->endOfSimulation ();}
   ...
};
```

Listing 2: *Helper class to ease access to the `sc_module` methods and attributes.*

```
%feature("director") sc_module_swig;
%{
 #include "systemc.h"
%}
%include "systemc/kernel/sc_module.h"
```

Listing 3: *Use of the director keyword to wrap correctly the `sc_module_swig` class.*

`SC_METHOD` or `SC_THREAD` macros. We show the implementation of the `sc_python_callback` class in the Listing 4. We have overridden the `run` method that executes the associated Python call-back.

We have added two methods to the `sc_module_swig` to support the `SC_METHOD` and `SC_THREAD` macros. Actually, we have created respectively the `method` and `thread` methods. We show in the Listing 5 the implementation of the `method` method: This method is wrapped and available in the script side. Thus, it is possible to create `SC_METHOD` and `SC_THREAD` in Python. The implementation of this method creates in line 2 a `sc_python_callback` object and in line 11 it registers the created callback to the SystemC engine. Next, we add the new `sc_method_handle` to all sensitive objects.

Other modifications were done on SystemC to make all SystemC functionalities available in

```
class sc_python_callback :
       public sc_process_host{
 public:
  sc_python_callback (PyObject *c) :
    callable (c) {Py_INCREF(callable);}

  ~sc_python_callback ()
    {Py_DECREF(callable);}

  void run () {
    PyObject * pyres;
    pyres = PyObject_CallObject(callable, NULL);
    Py_XDECREF(pyres);
  }

 private:
  PyObject * callable;
};
```

Listing 4: *Creation of `sc_python_callback` to allow Python process in the SystemC simulator engine.*

```
void sc_module_swig::method (PyObject * f){
  sc_python_callback * callback =
    new sc_python_callback (f);
  PyObject* name = PyObject_GetAttrString (f,
                               "__name__");
  if(name == NULL || !PyString_Check(name)){
    std::cerr << "python name error\n";
    return;
  }
  sc_method_handle handle = simcontext()->
    register_method_process(
      PyString_AsString (name), callback,
      this );
  sc_module::sensitive << handle;
  sc_module::sensitive_pos << handle;
  sc_module::sensitive_neg << handle;
}
```

Listing 5: *Implementation of the `SC_METHOD` helper to create `SC_METHOD` in the script side.*

Python. Moreover, we have added some functionalities to add and remove dynamically modules and connections from the simulation engine. Another added functionality in SystemPy is the capability to clean the simulation engine. It allows us to re-initialize the SystemC engine and load another design in the Python interpreter without restarting it.

4 Experimental Results

In this section, we show how one can create a design in Python using modules written in SystemC and modules written in Python. Then, we explain how to dynamically replace a module by another one. Finally, we generate a benchmark to compare languages performances.

4.1 Mixed SystemC/Python Example

Our example is a simple design composed of four IPs, two data generators, an adder and an IP that displays the results. In Figure 5 we show our design. The `Generator` and the `Display` modules are written in SystemC whereas the `Adder` module is written in Python. The Listing 6 shows the `Adder` module implementation. The creation of a SystemC module is done by

```
class Adder (sc_module_swig):
  def __init__ (self, name):
    sc_module_swig.__init__ (self, name)
    self.in_a = sc_in_int()
    self.in_b = sc_in_int()
    self.out = sc_out_int()
    self.method(self.run)
    self.getSensitive()<<self.in_a<<self.in_b

  def run (self):
    self.out.write(self.in_a.read ()
                   + self.in_b.read ())
```

Listing 6: *Example of a module implemented with the SystemC Python binding.*

inheriting from the `sc_module_swig` class. In the constructor we define input and output ports as well as the `run` method as a `SC_METHOD` process. After, we add sensitive entries. This module performs an addition of the two data entries and writes the result to the out port.

We show the importation and instantiation of SystemC and Python modules in the Python Listing 7. After this step, one can start the simulation with the `sc_start` function.

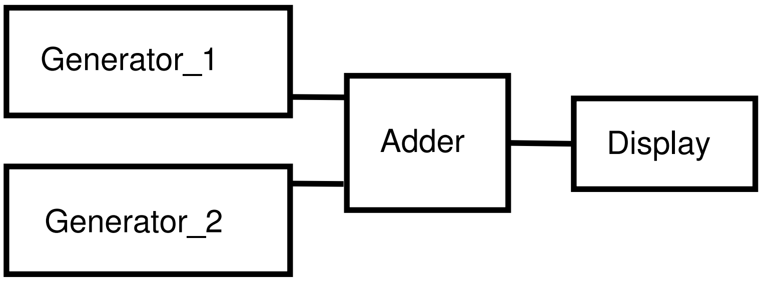


Figure 5: *Example of a design, Generators and Display are written in SystemC while the Adder is written in Python, demonstrating the flexibility and interoperability of our approach.*

4.2 Replacing a Module Dynamically

The dynamic replacement of a module is not possible in the current SystemC version. We had to make it possible modifying the SystemC kernel. Thus, it is now possible to suspend a simulation and to replace a module by another one (Listing 8). We have created a `Multiplier` module

```python
from systemc import *
from display import Display
from generate import Generate
from add import Adder

gen1 = Generate("gen1")
gen2 = Generate("gen2")
add1 = Adder ("add1")
disp1 = Display("display1")

s1 = sc_signal_int("s1")
s2 = sc_signal_int("s2")
s3 = sc_signal_int("s3")
s4 = sc_signal_int("s4")

gen1.out.bind(s1)
gen2.out.bind(s2)
add1.in_a.bind (s1)
add1.in_b.bind (s2)
add1.out.bind (s3)
disp1.in_a (s3)
```

Listing 7: *A toplevel written with the SystemC Python binding.*

```python
sc_disconnect_module(add1)
mul1 = Multiplier("mul")
mul1.in_a.bind (s1)
mul1.in_b.bind (s2)
mul1.out.bind (s3)
sc_resume ()
```

Listing 8: *End user can suspend the simulation to replace some modules.*

that replace the `Adder` module. This replace operation is done in three steps. The first one is the disconnection of the `Adder` module with the help of the `sc_disconnect_module` function. Then, we instantiate the `Multiplier` module and finally we connect it to the system. After the replace operation, the simulation can be resumed. The dynamic replacement is not time consuming, so the simulation performance is not affected by the replacement mechanism. The overall simulation performance can only be affected by the performance of the new added module. One of the major interest in dynamic replacing of module is by example replacing a TLM module by an RTL module. If you have an RTL system using a processor, you can accelerate the boot sequence replacing your RTL processor by the same processor define in TLM. Hence, after the boot sequence is finished with the TLM processor, you may replace it using the RTL processor.

4.3 Benchmark

Three kind of tests have been done: the first is test written entirely in SystemC, the second is written entirely in Python and the third one is using SystemC module with the top level written in Python.

We show in Table 1 the simulation time of each test and the number of lines of code to realize the system according to the language. We can see that the best simulation time is reached by

	SystemC	**Python**	**Mixed**
Simulation time (%/SystemC)	5.420s (100%)	75.850s (1399.45%)	5.500s (101.48%)
number of lines (%/SystemC)	112 (100%)	63 (56.25%)	105 (93.75%)

Table 1: *Simulation time and number of lines for each language, percentages in parenthesis is in relation with the figures of the full SystemC implementation.*

the system written in SystemC, followed by the mixed version with a minor simulation time difference. The worst simulation time is obtained by the system written in Python. However, an interesting result is the number of lines required to realize the system. The number of lines required to develop the system in Python is twice less lines than SystemC version. The number of lines of the mixed version can be explained by the fact that all modules are written in SystemC and only the top level is written in Python.

5 Summary and future work

Combining a system level description language and a scripting language within the same framework for design can lower significantly the time to market of a design. Indeed, it allows a fast architecture exploration and simulation of complex systems. Our framework called SystemPy mixes SystemC and Python. It allows dynamic IPs changes for composing a system without stopping the simulation. The methodology integrated in SystemPy is composed by three major steps: IPs creation, system composition and simulation. Thus, we can obtain with SystemPy an executable composed of some SystemC parts and Python parts, communicating using SWIG interfaces. Different steps of our methodology are detailed on an application example containing four modules. By introducing scripts into this example, we obtained very interesting simulation time and a significant reduction of the code size representing the system.

References

[Amo00] Guido Amount. SystemC standard. In *Proc. of ASP-DAC 2000*, pages pp.573–577, Yokohama, Japan, January 2000.

[BFSVT00] M. Baleani, A. Ferrari, A. L. Sangiovanni-Vincentelli, and C. Turchetti. Hw/sw codesign of an engine management system. In *DATE 2000*, pages 232–237, Paris, France, March 2000.

[DSG03] Frederic Doucet, Sandeep Shukla, and Rajesh Gupta. Introspection in system-level language frameworks: Meta-level vs. integrated. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10382. IEEE Computer Society, 2003.

[DSGO02] F. Doucet, S. Shukla, R. Gupta, and M. Otsuka. An environment for dynamic component composition for efficient co-design. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 736. IEEE Computer Society, 2002.

[Ini] The Open SystemC Initiative. http://www.systemc.org.

[Kni] Steven Knight. Scons - a software construction tool. *http://www.scons.org.*

[Kru98] Philippe Kruchten. Modeling component systems with the unified modeling language. 1998.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, March 1998.

[Pan01] Preeti Ranjan Panda. SystemC: a modeling plateform supporting multiple design abstraction. In *ISSS*, pages 75–80, 2001.

[Pre00] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, October 2000.

[Rig04] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM*, pages 15–26, 2004.