ReSP: A Non-Intrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration

Giovanni Beltrame Cristiana Bolchini Luca Fossati Antonio Miele Donatella Sciuto

Europeran Space Agency
Noordwijk, NL 2200 AG
Giovanni.Beltrame@esa.int

Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milan, IT 20133
{bolchini,fossati,miele,sciuto}@elet.polimi.it

Abstract—This paper presents ReSP (*Re*flective *S*imulation *P*latform), a Transaction-Level multi-processor simulation platform based on SystemC and Python; SystemC is a standard language for system modeling and verification, and Python provides the platform with reflective capabilities. These are employed to give the designer an easy way to specify the architecture of a system, simulate the given configuration and perform automatic analysis on it. ReSP enables SystemC and Python interoperability through automatic Python wrapper generation. We show that the overhead associated with the Python intermediate layer is around 1%, therefore execution speed is not compromised. The advantages of our approach are: (a) easy integration of external IPs (b) fine grain control of the simulation (c) effortless integration of tools for system analysis and design space exploration. A case study shows how the platform can be extended to support system reliability assessment.

I. Introduction

Nowadays the design of embedded systems and System-On-Chip devices (SoC) is increasingly based on multiple processors leading to a new approach, called MultiProcessor System on Chip (MPSoC). What has been dubbed "MPSoC" is becoming the prevalent design style to achieve tight time-to-market design goals, to maximize design reuse, to simplify the verification process and to provide flexibility and programmability for post-fabrication reuse of complex platforms.

MPSoCs are composed of off-the-shelf processing cores, memories and application specific coprocessors. These Multi-Processor architectures introduce new difficulties in the development and design flow:

- The architecture may include more than one application specific master processor;
- Inter-processor communication may require more sophisticated networks than a simple shared bus;
- Processing elements cannot be independently designed and optimized: this causes an exponential growth of the design space.

These new challenges for system architects, software and hardware designers, verification specialists and system integrators may best be met by revisions to old tools, by using methods to deal with MPSoC complexities, by introducing new tools and methods working at the same abstraction levels and by moving up in abstraction to take advantage of new design approaches. Moreover, in order to obey to tight market constraints, the SoC design process must rely on pre-designed or third party components. Components obtained from different providers, and even those designed by different teams of the same company, may be heterogeneous on several aspects: design domains, interfaces, abstraction levels, granularities, etc. Therefore, component integration is required at system level.

In this paper we present *ReSP* (*Re*flective *S*imulation *P*latform), an MPSoC simulation platform working at a high abstraction level. Our simulation platform uses a component based design methodology [6]: it provides primitives to build complex architectures from basic components. This bottom-up approach allows design-architects to explore efficient custom solutions with best performance. Components used by ReSP are built on top of SystemC and TLM hardware and communication description libraries [2]. SystemC is a C++ library that provides hardware modeling concepts (e.g. time, concurrency, events, bit, types, etc.) for simulation of hardware/software systems at different levels of abstraction. Transaction Level Modeling (TLM) has been introduced in recent years as a modeling style to describe communication channels at a higher abstraction level with respect to Register Transfer Level.

The simulation platform is built using Python programming language; its *reflective* capabilities (see Sec. III for a definition of reflection) augment the platform with the observability of the internal structure of the SystemC components. This feature enables run-time composition and dynamic management of the architecture under analysis. The full potentialities offered by the integration among Python and SystemC are exploited, during simulation, to query, examine and, possibly, modify the internal status of the hardware models. These capabilities simplify the debugging and testing processes for both the modeled hardware architecture and the software running on it.

In conclusion, the advantages of coupling standard SystemC based simulation techniques together with the reflective capabilities of Python are multiple: (a) introspection inside the components, which simplifies the integration of external IPs, (b) fine grain control of the simulation, allowing an easy monitoring and modification of the status of the components, and (c) effortless integration of new tools for the analysis and the design space exploration; for instance, the application scenario

978-1-4244-1922-7/08/$25.00 ©2008 IEEE

presented in the paper shows how the simulator has been extended with features for system reliability assessment.

This paper is organized as follows: Section II presents the state of the art of hardware simulators describing advantages and limitations of the most common works found in literature; Section III describes the ReSP architecture in detail. Section IV discusses how ReSP can be used to evaluate the reliability level of a simple architecture. Conclusions are drawn in Section V.

II. Related Work

Simulation is the most commonly adopted technique for the analysis of functional and non-functional requirements of Multi-Processor platforms. A wide number of simulators have been proposed in literature; we now provide a brief overview of the ones that are relevant to this paper.

StepNP [12] is a platform for the exploration of Network Processor Units; the platform presents several limitations: it features a reduced set of Instruction Set Simulators (ISS) and, in order to integrate new ones, ad-hoc wrappers in SystemC have to be created. Moreover TLM standard library is not used for communication. In our platform the need for ad-hoc wrappers has been eliminated thanks to the integration among C++ and Python and, consequently, the introduction of reflective properties.

Beltrame et al. [5] extended StepNP introducing the concept of *introspection* to support dynamic switching of simulation accuracy; to augment SystemC with reflective capabilities they created SIDL, a CORBA like interface definition language: each component has to be modified *manually* to extend a specific class that provides functionalities to let external processes read or modify part of the component status. Even if the approach is innovative, it is considerably intrusive and does not allow complete control over the component model and its internal state.

Another simulator proposed in literature is Platform Designer [3], a framework of SystemC tools that provides support for modeling, simulation and analysis of multi-processor systems; they are modeled as an extension of processor constructs. This work lacks support for custom buses (only OCP and AMBA are currently supported) and it does not allow modeling of architectures different from the ones based on a shared bus; processors must also have a private memory region. Finally, communication through TLM interfaces is not supported. In addition this system is completely based on C++: as consequence it has no support for reflection and a limited flexibility.

CoWare Platform Architect [8] uses SystemC to model and simulate the platform; it also integrates a Processor Designer for the specification of ISSs. While CoWare Platform Architect contains a wide library of components, integration of custom models requires a considerable effort.

Integration among Python and SystemC has been explored in [16] to test an RF CMOS transreceiver; a standard HDL simulator was not satisfactory in that it worked at a too low abstraction level and it did not offer enough flexibility. The integration was achieved through Inter Process Communication (IPC). In this work we do not focus on using Python as a driver for the simulator, but we try to embed Python inside the simulator itself.

The work presented in [17] proposes another way of integrating Python and SystemC: Python is used to embed scripting into SystemC modules. This approach, however, requires the modification of the OSCI SystemC kernel, and it cannot provide direct access to private and protected methods in Python. Embedding Python proves to be effective in reducing the number of lines of code used to express a given functionality, at the expense of an order of magnitude of simulation speed reduction. In this work, we do not focus on the embedding of scripts in SystemC, but on the use of Python features and on the automatic wrapper generation for SystemC modules, and we overcome the issues of [17] by using a different wrapping approach.

At the best of the authors' knowledge, ReSP provides the most complete Python wrapping mechanism for SystemC and TLM designs.

III. Simulator Infrastructure

This work presents ReSP, a Reflective Simulation Platform. The idea is to give the designer an easy way to specify the architecture of a system, simulate the given configuration and perform automatic analysis (such as design space exploration or reliability assessment) on it. To achieve this goal, ReSP starts from SystemC and the OSCI Transaction Level Modeling library [18] and provides a non-intrusive framework to manipulate SystemC and TLM objects. This work is particularly suitable for platform-based design: the use of a well-defined set of architectural elements and the design space exploration on the interconnection, number and parameters of those elements, are keys for the effectiveness of the design methodology. In the following, the terms *platform* and *framework* will be used to indicate the overall ReSP architecture.

ReSP uses a formalism to describe the components and the interconnections between components of a system, as many Architecture Description Languages (ADLs) do. Components are chosen from a database of SystemC modules. In the following, we will refer to the term *component* to describe any top-level SystemC module included into the framework's database. The proposed framework is based on the concept of reflection [9], that allows ReSP to view and modify every C++ or SystemC

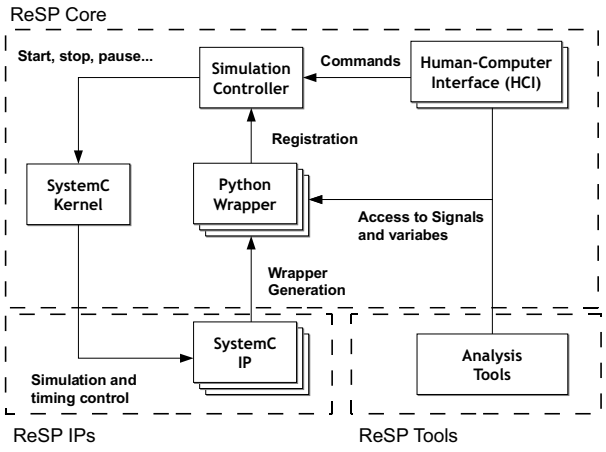

Fig. 1. The architecture of the ReSP system simulation platform

element (variable, method, etc.) specified in any component. In [15] we read that "reflective computational systems allow computations to observe and modify properties of their own behavior, especially properties that are typically observed only from some external, meta-level viewpoint"; this means that the program itself is aware of its structure thus, for example, the simulator can inspect a component and get all its communication ports without the need to have any a-priori knowledge on the component's methods and variables. This concept was introduced in [5], but required the use of an interface description language (IDL) to describe IP capabilities and therefore some extra effort to include the IP in the framework database.

ReSP removes the need for special Interface Definition Languages (IDL) for specifying interfaces; SystemC code is directly parsed and the interface files (here called *wrappers*) are automatically generated. This means that standard SystemC TLM IPs can be integrated in the system with minimum effort.

As Fig. 1 shows, ReSP is composed of three main parts: ReSP *Core*, *IPs* and *Tools*.

A. ReSP Core

The core of the ReSP architecture is the OSCI standard SystemC kernel, as directly released by OSCI. This is an advantage when compared to other works [17], as they require modifications to the SystemC kernel. ReSP provides a wrapper for the Python scripting language around the SystemC kernel. Python inherently supports reflection, and allows access to SystemC variables and arbitrary function calls to SystemC. The *Simulation Controller* is a set of Python classes that translate commands coming from the user into SystemC function calls, controlling the simulation behavior. As an example, it is possible to run, step, pause, or stop the simulation at runtime, something not present natively in SystemC. This concept was introduced in [12] and it is now widely used.

The User Interface (or Human Computer Interface, HCI) is also written in Python and it represents an interface between the simulation controller and the user. This architecture allows multiple interfaces (such as command line or graphical ones) to be built.

The novelty introduced by ReSP lies in the Python wrapper generation for SystemC and TLM components. In previous works [12, 17, 5, 3, 8], the developer had to write a special interface file or use some specific classes in order to add a component to the framework's database; moreover only the components' characteristics described in those interface files could be used by the simulator. ReSP deals with this step automatically, by generating the Python wrapper right after parsing the component C++ header file. The generation flow is shown in Fig. 2.

Each header file is parsed using GCCXML, a tool that provides an XML description of the GCC abstract syntax tree. The resulting XML description is manipulated to select all the parts that need to be exported, and then the opensource tool py++ is used to generate Python wrapping code that uses the Boost.Python library [1]. The advantage of Boost.Python and py++ over alternative tools like SWIG (used by most other works) is that it guarantees access to all C++ declarations, even private or protected ones, through the generation of appropriate class wrappers. The Python interpreter can load the extensions generated by the ReSP flow, and have *full* access to the C++, and therefore SystemC, classes contained in the exported module. Another feature of the ReSP flow is that IP documentation is automatically extracted from the SystemC source code, and inserted in the Python wrapper. Python self-documentation features are then used to display such documentation through the User Interface.

The SystemC kernel and the Simulation Controller are run in one execution thread with the rest of the Python wrappers. HCI and tool access are executed in a separate thread, and synchronized with the SystemC kernel. In this way, the user has full asynchronous control of the simulation (the status of the components can be queried and/or modified even while simulation is running), without consistency loss.

B. ReSP IPs

One of the peculiarities of ReSP is the capability of integrating any valid SystemC component in an easy way; in fact, as described in the previous section, it is not necessary to modify components' descriptions due to the fact that ReSP automatically generates the Python wrapper. This favors external IP reusability and the description of new hardware architectures by composition of already existing components. Currently, the simulation platform includes the following component models:

- *processors cores* written using the ArchC [14] Architectural Description Language; we possess both the functional and cycle accurate versions of the PowerPC, Leon2 and ARM7 RISC processors;
- *interconnections* in terms of bus and Networks-On-Chip;
- *memory systems* including simple memories and caches;
- *miscellaneous* components, such as UARTs and interrupt controllers.

Additional components can be easily added by putting their SystemC source code in the ReSP build tree. No additional interface or glue code needs to be written, as ReSP automatically generates the appropriate component wrappers.

C. ReSP Tools

The introduction of reflection paves the way for the development of a set of tools to perform system-level analysis. Any operation that requires observability, can be performed through the Python wrappers. For example, it is possible to include advanced network traffic analysis (latency, throughput. etc.) by

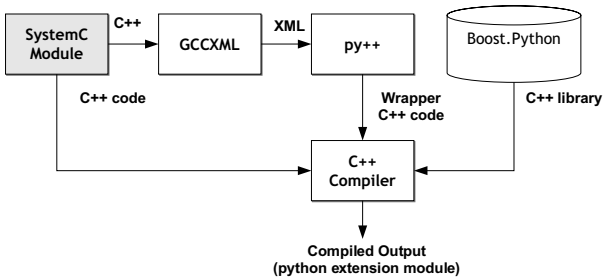

Fig. 2. ReSP wrapper generation flow

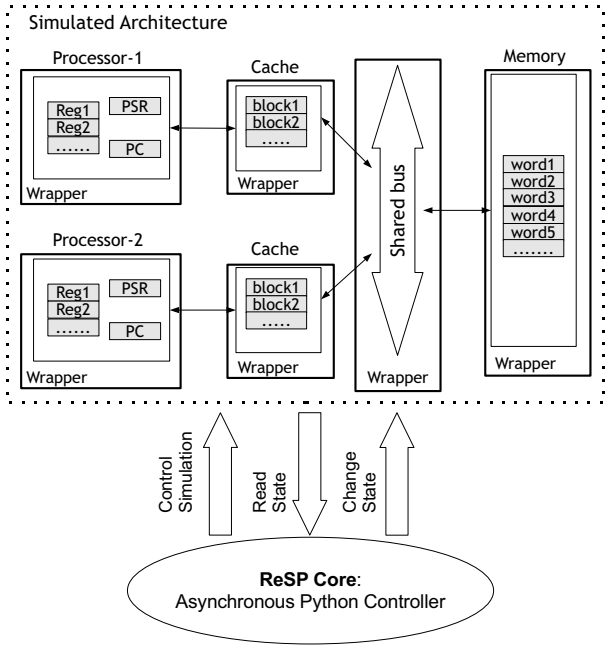

Fig. 3. Components of the ReSP platform: the architecture is composed of SystemC components together with the corresponding automatically created Python wrappers; the ReSP core is used to manage the SystemC kernel and the conponent models.

observing the network traffic, or add power modeling to the system by extracting switching activities from the system at runtime. The biggest advantage given by the use of Python lies in the decoupling among the simulator itself and the SytemC models; the simulator does not need to have any a-priori knowledge about the components' structure: there is not need to change the simulator's code even if some components are modified.

Two execution modes are available: interactive and automatic. The first one allows step-by-step execution of the architecture under analysis. The architecture can be built using the commands exported by the User Interface: components are seen as normal Python classes which are instantiated and connected together executing standard Python commands. Automatic instantiation, by means of an XML file, is also possible. Interactive execution mode helps the designer in having a deeper insight on the modeled architecture; this mode is especially useful during debug activity which, if the reflective capabilities of the platform are also used, can be performed in a very efficient way.

Automatic execution mode is used to run in batch mode a sequence of n simulations. When using automatic mode, the architecture description is provided in an XML file. The usefulness of this mode emerges when many runs of the same architecture have to be performed; this is, for example, the case with design space exploration algorithms (only component parameters, such as the number of processors, the cache size etc. change over different runs) or with fault injection campaigns (different faults are injected over different runs). Section IV shows how ReSP can be used to perform reliability analysis using its tool interface.

Example Let us suppose that we need to load the architecture described in Fig. 3 in order to debug it and the software running on it (Listing 1 shows commands for loading the architecture). We detect a problem in the architecture after the execution of n clock cycles: simulation can be paused (by simply issuing the Python command `controller.pause_simulation()`) and, through the User Interface, the internal status of the component models (i.e. the value of all their variables) can be examined and, if necessary, modified:

```
proc1.PSR.read()
proc1.PSR.write(15)

proc2.PSR.read()
proc2.PSR.write(45)
```

This greatly helps in identifying, for example, which processor register contains an erroneous value. Normally such a debug activity would require consistent changes in the ISS in order, at least, to integrate it with a software debugger, such as GNU/GDB, while in ReSP, thanks to the reflective facilities offered by Python, no effort is required by the components' developer. ReSP debugging capabilities are also enhanced by the presence of callbacks: actions can be associated with specified conditions; for example, when the program counter assumes a specified value we can automatically pause simulation.

Listing 1 : Python commands used to load the architecture of Fig. 3

```
proc1 = leon2.leon2('proc1')
proc2 = leon2.leon2('proc2')

mem = SimpleMemory32.SimpleMemory32( 'mem' )
mem.setSize(1024*1024*8)

bus = pv_router32.pv_router32('SimpleBus')

manager.connectPortsForce(proc1, 'proc1', proc1.
    DATA_MEM_port.memory_port, bus, 'SimpleBus', bus.
    target_port[0])
manager.connectPortsForce(proc1, 'proc1', proc1.
    PROG_MEM_port.memory_port, bus, 'SimpleBus', bus.
    target_port[0])
manager.connectPortsForce(proc2, 'proc2', proc2.
    DATA_MEM_port.memory_port, bus, 'SimpleBus', bus.
    target_port[1])
manager.connectPortsForce(proc2, 'proc2', proc2.
    PROG_MEM_port.memory_port, bus, 'SimpleBus', bus.
    target_port[1])

manager.connectPortsForce(bus, 'SimpleBus', bus.
    initiator_port, mem, 'mem', mem.memPort)
```

D. ReSP Performance

Fig. 4 shows the speed-up obtained by native SystemC execution with respect to the execution of the same component models inside ReSP. The first experiment (Fig. 5 on the left) was set-up to measure the transactional speed of the system; it consisted in the connection of basic master and slave components: the former sends characters to the latter component. On the right there are the results of the execution of a full architecture; this was created by connecting a functional Leon2 processor model (created using the ArchC [14] architectural description language) and the TLM Programmer's View (PV) memory and bus. The number of instructions per second, obtained both using native execution and execution inside ReSP, are shown.

From Fig. 4 it is clear that the small performance penalty due to the additional software layer introduced by Python is negligible, especially if the advantages coming with to the introduction of Python are considered.

All the experiments were hosted on a 2 GHz Intel Core 2 Duo System with 2 GB of RAM running Gentoo Linux.

IV. Application Scenario: Fault Injection

The reflective capabilities provided in ReSP can be used also for other purposes besides architecture composition and its dynamic management; in particular, we have exploited these features to implement a fault injection environment. We have followed the SoftWare-Implemented Hardware Fault Injection (SWIFI) [4] approach, based on the modification of the components' internal state and on the simulation of the system behavior in presence of a hardware failure.

Works proposed in literature pursue fault injection by means of code instrumentation for accessing the internal state of the architecture [10]. By exploiting reflection instrumentation is not necessary and, therefore, it is possible to perform fault analysis (a) in a transparent way and (b) significantly reducing the set-up time necessary to be able to carry out the experiments. Only a few works have exploited reflective programming [11], devising, however, solutions strictly related to reliability assessment. In our case, the adoption of SystemC and TLM allows us to propose a flexible framework which is quite innovative also w.r.t. fault injection scenarios.

The possibility of modifying the internal state of the components allows the adoption of a generic functional fault model as well as a radiation induced fault model, such as Single/Multiple Event Upsets. In this work the considered fault model is the soft error or Single Event Effect that represents a transient misbehavior mainly caused by radiations; this behavior affects the devices causing a bit-flip of a value stored in a memory cell. The classes of faults that can be simulated strictly depend on the abstraction level adopted by the component description. At present the components available in ReSP are described at a functional level, hence, the injected faults can only be modeled at behavioral level, rather than structural one; a common approach when considering the complexity of the described cores. At the same time, if a structural description of a component were available, a low abstraction level fault (such as a stuck-at) might be modeled and dealt with.

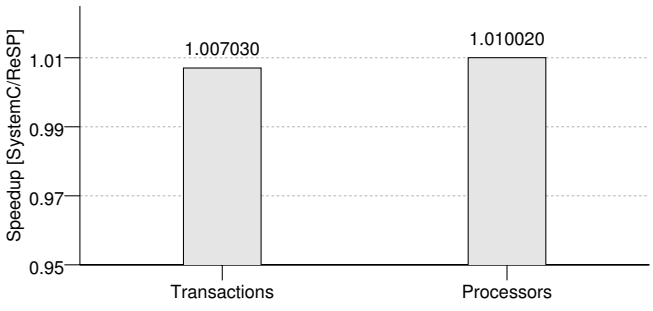

Fig. 4. Speedup of native execution with respect to execution inside ReSP.

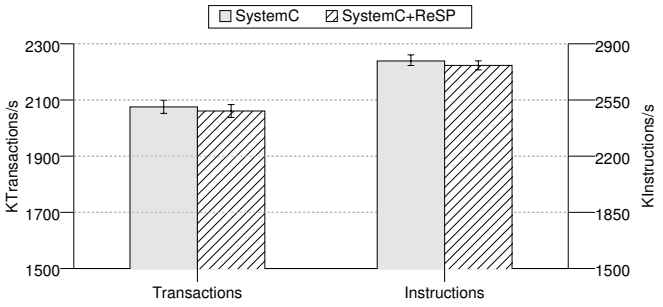

Fig. 5. Execution Speed of a both of generic hardware architecture (on the left) and a processor/bus/memory architecture (on the right) measured first using plain SystemC and then using ReSP.

Moreover, ReSP provides the possibility to automatically instantiate the golden model, i. e., a copy of the system under test used for comparing the faulty system behavior with a fault-free one.

Our fault injection environment can use both the execution modes described in Sec. C; during the interactive experimental session, simulation is executed step-by-step; the user can run the simulation and manually suspend it in order to inject faults in the desired storage location. Then, she/he can resume the simulation to monitor the internal state of the architecture under test and to compare it with the golden model (i.e., a fault-free copy of the system under test) in order to analyze the failure evolution. On the other hand, automatic execution can be employed to realize automated fault injection campaigns. The list of faults to be injected is specified through an XML file: each fault is identified in terms of the component and the variable to be changed, the mask to be applied for changing the variable value and the clock cycle at which injection has to be performed. When the experimental session is launched, simulations, one for each element in the list, are performed. The final report, stored in a file, shows for each simulation if the fault has been activated and if it has been detected by fault tolerance features of the circuit under test.

We have used ReSP for reproducing the experimental session proposed in [13]. The purpose of that case study is to evaluate the capabilities of software redundant techniques in detecting faults affecting microprocessors: the system under test is a Leon2 processor running three different applications hardened with software redundant techniques; the initial fault injection environment [7] consisted of an FPGA board emulating the instrumented model of the processor. The same fault injection campaign was repeated by using ReSP: we specified a simple architecture composed of a Leon2 functional model connected to a memory through a bus and we ran the same software applications used in the previous experiment. Several processor registers (e.g.: the Program Counter register, the register bank, the Y register and the PSR register) were indicated as possible fault locations. Table I presents the results of our experimental session ("HW Detected" column reports cases where an interrupt has been asserted because of an error detection or because of a pre-defined time out expiration).

Our approach shows several advantages with respect to the one adopted in [13]: the capability of performing fault injec-

TABLE I
RELIABILITY ANALYSIS: EXPERIMENTAL RESULTS

Application	Register	Faults	No Error	Error		
				HW Detected	SW Detected	Not detected
ELPF	Reg. Bank	2000	1787	51	152	10
	PC Reg.	1000	775	12	207	6
	Other Regs	600	591	0	9	0
FIR	Reg. Bank	2000	1742	85	154	19
	PC Reg.	1000	663	93	235	9
	Other Regs	600	571	0	27	2
Kalman	Reg. Bank	2000	1540	185	271	4
	PC Reg.	1000	591	62	346	1
	Other Regs	600	593	0	7	0
TOTAL		10800	8853	488	1408	51

tion by means of introspection allows to carry out experiments in a faster and transparent way (i.e. no modifications to the processor code are needed). It is worth noting that setting up the experimental environment and executing the whole fault injection campaign took only one hour, while instrumenting the processor description for the experiment proposed in [13] took several days. Moreover, our approach does not require complex devices such as FPGAs. Finally, we can perform fault injection experiments at several abstraction levels simply by changing the abstraction level of the components plugged into ReSP.

V. SUMMARY AND CONCLUSIONS

In this paper we presented ReSP, a hardware simulation platform targeted to Multi-Processor Systems-On-Chip; the platform is based on the integration of Python and SystemC allowing effortless integration of external IPs and custom components. Python augments ReSP with reflective capabilities enabling a fine grained control over simulation and over the internal status of the component modules; this offers advantages, with respect to traditional simulators, in the tasks of reliability analysis (as shown in Sec. IV), design space exploration and debug and test of the hardware/software system under analysis. Results show that integration among Python and SystemC does not introduce significant overhead over plain SystemC and C++ execution. The effectiveness of our approach was presented through a case study on the software reliability in presence of hardware failures.

REFERENCES

[1] C++/python interfacing: pyplusplus. `http://www.language-binding.net`.

[2] Open SystemC Initiative: `http://www.systemc.org`.

[3] C. Araujo, M. Gomes, E. Barros, S. Rigo, R. Azevedo, and G. Araujo. Platform designer: An approach for modeling multiprocessor platforms based on SystemC. *Design Automation for Embedded Systems*, Vol. 10:253–283, 2007.

[4] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Günther H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Trans. Comput.*, 52(9):1115–1133, 2003.

[5] G. Beltrame, D. Sciuto, C. Silvano, D. Lyonnard, and C. Pilkington. Exploiting TLM and object introspection for system-level simulation. In *DATE '06: Proc. of the conference on Design, Automation and Test in Europe*, pages 100–105, 2006.

[6] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore SoCs. In *DAC '02: Proc. of the 39th conference on Design automation*, pages 789–794, 2002.

[7] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante. An FPGA-Based approach for speeding-up fault injection campaigns on safety-critical circuits. *Journal of Electronic Testing: Theory and Applications*, 18(3):261–271, 2002.

[8] CoWare. CoWare Platformm Architect. `http://www.coware.com/`.

[9] B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 327–336, 1989.

[10] M. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[11] E. Martins, C. M. F. Rubira, and N. G. M. Leme. Jaca: A reflective fault injection tool based on patterns. In *Proc. on Dependable Systems and Networks*, pages 483–482, 2002.

[12] P.G. Paulin, C. Pilkington, and E. Bensoudane. StepNP: A System–Level Exploration Platform for Network Processors. *IEEE Design and Test of Computers*, pages 2–11, November–December 2002.

[13] M. Rebaudengo, L. Sterpone, M. Violante, C. Bolchini, A. Miele, and D. Sciuto. Combined software and hardware techniques for the design of reliable ip processors. In *Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, DFT*, pages 265–273, 2006.

[14] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: A SystemC-Based Architecture Description Language. *sbac-pad*, 00:66–73, 2004.

[15] J. Sobel and D. Friedman. An introduction to reflection-oriented programming, 1996.

[16] N. Tribie, O. Fargant, and S. Antipolis. A Python Based SoC Validation and Test Environment. *Design & Reuse Industry Articles*.

[17] J. Vennin, S. Penain, L. Charest, S. Meftali, and J. Dekeyser. Embed scripting inside SystemC. In *Forum on Specification and Design Languages, FDL'05*, 2005.

[18] L. Yu, S. Abdi, and D. Gajski. Transaction Level Platform Modeling in SystemC for Multi-Processor Designs. Technical report, Center for Embedded Computer Systems, University of California, Irvine, 2007.