From VHDL Register Transfer Level to SystemC Transaction Level Modeling: a Comparative Case Study

Ney Calazans, Edson Moreno, Fabiano Hessel, Vitor Rosa, Fernando Moraes, Everton Carara
Pontifícia Universidade Católica do Rio Grande do Sul (FACIN-PUCRS)
Av. Ipiranga, 6681 - Prédio 30 / BLOCO 4 - 90619-900 - Porto Alegre – RS – BRASIL
{calazans, emoreno, hessel, vitormr, moraes, carara}@inf.pucrs.br

Abstract

Transaction level (TL) modeling is regarded today as the next step in the direction of complex integrated circuits and systems design entry. This means that as this modeling level definition evolves, automated synthesis tools will increasingly support it, allowing design capture to start at a higher abstraction level than today. This work presents a comparison of traditional register transfer level (RTL) modeling and transaction level modeling through the implementation of a simple processor case study. SystemC is a language that naturally supports hardware transaction level descriptions. The R8 processor was described in SystemC TL and RTL versions and these were compared to an equivalent hand-coded VHDL RTL description in some key points, such as simulation efficiency and implementation results. The experiments indicate that TL descriptions present a faster path to system validation and that it is possible to envisage the automation of the design flow from this level of abstraction without significant impact on the quality of the final implementation.

***Key words**: System modeling, register transfer level, transaction level, SystemC, VHDL.*

1. Introduction

The current mainstream technology employed to capture the design of a complex integrated circuit (IC) is based upon the use of hardware description languages (HDLs) such as VHDL or Verilog. These languages support the description of circuits at a range of abstraction levels varying from gate level netlists up to purely algorithmic behaviors. However, HDL design capture traditionally starts with an informal specification, generating from it a detailed, clock cycle accurate hardware description. As the complexity of ICs increase, the direct passage from informal product specifications to clock cycle level accurate HDL descriptions becomes less feasible, because the amount of information to aggregate to a specification becomes unmanageable.

Today, designing application specific ICs (ASICs) with tens of millions of transistors is commonplace. Such ICs are complex enough to contain all major elements of a complete end product, and are accordingly known as Systems-on-Chip (SoCs) [1]. A SoC usually contains one or more programmable processors, on-chip memory, peripheral devices, and specifically designed complex hardware modules. Also, according to the SIA Roadmap [2], somewhere between years 2010 and 2013, a state of the art ASIC will be allowed to contain more than one billion transistors. Capturing the design of current and future SoCs requires enhancements in current design practice. These enhancements include both new classes of *resources* to use in the design process and new classes of *design techniques* to employ.

Essential resources of every SoC are *intellectual property cores* (IP cores) [3][4]. An IP core is a complex module that fulfills some specific task and is created for reuse. IP cores are the basic building blocks of SoCs. Previsions are that by 2015, IP cores will compose 90% of the area of a SoC [2]. Hardware modules inside a SoC communicate with each other using on-chip shared buses. Nonetheless, buses are not scalable, and can hardly cope with future global chip clocking strategies, currently migrating from a globally synchronous approach to a globally asynchronous, locally synchronous approach. To overcome these limitations, concepts used in computer networks, telecom and distributed systems are being adapted to the on-chip domain, creating a new kind of resource, usually called *Networks-on-Chip* (NoCs) [5][6].

Plugging hardware IP cores into a SoC design must be an effortless task. The proposition of *standard interfaces* like VSI [7] and OCP [8] provide core-centric interfaces also called *sockets*. Standard interfaces or sockets improve design reuse and scalability in SoC design with regard to interconnection-centric approaches provided by, e.g. standard buses. Another new resource for SoCs is reconfigurable hardware [9], which has the potential to greatly increase performance and flexibility. Adding reconfigurable resources to SoCs can provide hardware tuning capabilities and hardware revision to systems, improving product performance and lifetime. All the cited new resources may be combined into *platforms* [10], a resource to enable timely delivery of SoCs. Platforms are collections of hardware and software IP cores, together with on-chip communication structures designed to serve as a template of a SoC for a given application area, such as wireless communication or automotive applications.

Given an adequate platform, a specific SoC design may be limited to platform tuning and software adaptation.

As for the evolution of design techniques, since hardware and software compose SoCs, their design must rely on *hardware/software codesign* techniques [11]. This is mandatory because the separate implementation of these entities can no longer achieve the best performance and time-to-market figures for highly complex ICs. Also, to achieve design reuse in general and IP core reuse in particular, modules have to be designed *for reuse* [12]. Many of the design techniques common in the traditional ASIC design flow have to be revised or even discarded if a reuse method is strictly followed. Finally, it is of paramount importance to increase the *abstraction level* in which formal descriptions are used for design capture and automated processing both for synthesis and verification [10]. From design capture starting with physical descriptions of the chip layout, technology moved to structural schematics description, and from there to mixed structural-behavioral descriptions in HDLs. Each design capture generation brought important productivity gains for designers with regard to the previous generation. Among other advantages, HDLs and associated design tools have ascertained portability and cell library independence across design tools vendors. This increases the potential for reuse and for the widespread availability of IP cores in the market.

However, current SoC design is pushing HDLs to the limit. Automatically synthesizable HDL descriptions must contain too many details to allow global SoC validation by means of HDL simulators. Also, since there is no consensus as to which HDL is best, tools must support Verilog, VHDL and mixed designs, causing library incompatibilities and leading to inefficient tools [13]. Still worse, the HDL support for validation is poor when compared to software languages such as C/C++. Complex hardware requires complex validation processes that are reusable at several levels of abstraction.

This work addresses the evolution of design capture techniques in the SoC design context. One important contribution is to certify that TL modeling can already be considered better than RTL modeling for validation purposes, since it enables simulations an order of magnitude faster than HDL. Another connected contribution is an indication that SystemC may be a good candidate language to provide a smooth transition from TL to RTL modeling without hampering the quality of the design. This occurs because SystemC RTL modeling is capable of delivering hardware quality comparable to hand-coded HDL design.

The rest of this paper is organized as follows. Section 2 presents a brief comparison of register transfer and transaction abstraction levels for design capture. Also, it shows a transaction level design flow employed in the design of the case study described in Section 3. The SystemC implementations of the case study are the subject of Section 4. Section 5 provides initial results of the comparison for three different implementations of the case study. Section 6 presents conclusions and future work.

2. Design abstraction and flow

The register transfer level (RTL) of abstraction has been formally described [13]. Informally, RTL consists in describing hardware modules by means of a datapath and a control unit. The datapath is composed structurally, using a set of memory elements (registers), processing elements (ALUs, multipliers, etc) and interconnection between these. The control unit is a behavior description of how data flows and is transformed in the datapath. RTL modeling assumes clock cycle level accuracy in the datapath and control unit descriptions. Presently, synthesis tools accept RTL models as input for producing high-quality designs.

Abstraction levels beyond the RTL have been collectively known as *system levels*. There is no consensus on how such levels are composed, but there is today an agreement that they comprise more than one abstraction level. It is possible to find propositions of two [14], three [15] or even four [16] levels of abstraction above RTL. All of these agree that above the RTL level there is the *transaction level* (TL) of abstraction, the target level for this work. TL modeling is timed but not clock cycle accurate [16]. The first challenge to employ TL modeling for RTL designers is to abstract the clock and specific protocols. TL modeling is considered important for architectural performance analysis, hardware/software partitioning, and golden test patterns generation for several abstraction levels, among other uses. Using TL modeling has the potential to reduce design effort in other abstraction levels and increase simulation speed.

The first step in the TL design flow, *design capture*, consists in describing the system as a set of high-level components. Components are modeled as hierarchical modules that contain processes, ports, and abstract channels. Processes define the behavior of a particular module and provide a method for expressing concurrency. The communication between the modules (components) is performed through abstract channels. An abstract channel implements one or more interfaces, where an interface is simply a collection of method definitions (in the sense of object oriented languages). Abstract channels encapsulate abstract communication protocols. A process accesses a channel interface via a port on the module.

The *validation* of a TL design is done using a TL simulation. Test patterns used at this level are reused at subsequent design levels. Once system validation is achieved, the TL description needs to be *partitioned* into hardware and software. This may be performed manually or automatically using codesign tools. The presented case study does not include this partitioning step, being completely implemented in hardware.

Next, the TL specification is *refined* into an RTL specification. Since no tools for automating this refinement are currently available, this step is performed manually. Intermodule communication is implemented with wires. At this level, cycle-accurate communication protocols are defined and cycle-accurate timing of actions is delineated. The validation at this level is done by RTL simulation, using the same gold test patterns developed for the TL modeling.

The next steps in the design flow are supposed to be executed automatically and include (1) RTL to logic level synthesis; (2) physical synthesis.

SystemC [17] is a language that enables the use of several abstraction levels, including RTL and upper levels such as TL and the *untimed level* [15]. This language is used here to provide a single development framework for TL and RTL modeling. Our RTL to logic level synthesis employs the following tools: Synopsys SystemC Compiler for translating SystemC RTL to VHDL and Leonardo Spectrum or Xilinx XST for logic synthesis. The physical synthesis step uses the Xilinx ISE tools to target the design to Virtex FPGAs.

Concerning the functional validation for the TL and RTL models, the Synopsys CoCentric System Studio design capture and simulation environment is used. The VHDL RTL design was validated with Modelsim and Active-HDL simulators.

3. Case study

The case study is a simple, 16-bit load-store processor, named R8. It presents a regular instruction format: all instructions have exactly the same size, occupying one 16-bit memory word each. The instruction contains the operation code and the specification of the operands, in case these exist. There are just a few addressing modes. This processor is a RISC-like machine, but still missing some characteristics so common in most RISC processors, such as *pipelines*. The main specific organizational characteristics of this multi-cycle processor are:

- Address and data are 16-bit wide;
- Memory addressing is performed on a word basis;
- Register bank with 16 general-purpose registers;
- 4 status flags: negative, zero, carry, and overflow;
- Instruction execution takes place in 2 to 4 clock cycles, i.e. the average clock per instruction (CPI) for any program executed is a number between 2 and 4.

The R8 processor has already been prototyped in hardware and is used today in wide range of projects inside the research group. The processor is available free of charge at http://www.inf.pucrs.br/~gaph/Projects/R8/R8%20Processor%20Core.html. In addition to the R8 IP core, software tools, and a full application are also there.

4. SystemC implementations

This Section describes the case study design in two abstraction levels using SystemC 2.0.1 [17]. The first design is a TL modeling and the second one is an RTL modeling, which can be obtained by the refinement of the first one, as a top-down modeling approach.

4.1. Transaction level modeling

At this abstraction level, the focus is not on the processor-memory interface precise description or on the number and nature of the available registers. The focus is in the system functionality and on the external and internal communication, in an abstract way [18]. This enables early design analysis to find architecture bottlenecks that can compromise the efficiency of the system. To achieve this, modules describe the architecture functional components and channels are responsible for the communication between modules. Channels abstract the complexity of the protocols used, allowing the transmission of any desired data type, from a simple bit to a complex structure [14].

The case study was implemented with three modules and three channels, as depicted in Figure 1. The modules describe the behavior of the processor, the memory, and the register bank. Each channel describes the communication between the processor and the memory, the register bank and the flags, one for each.

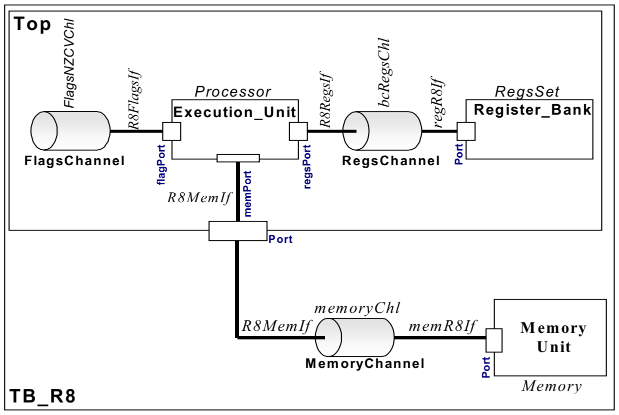


Figure 1 - R8 processor implemented with 3 modules (Execution Unit, Register Bank and Memory Unit) and 3 channels (FlagsChannel, RegsChannel and Memory Channel). Boldface names are instances others are classes.

Based on the specification of the case study [19], the processor is modeled to fetch the instructions from memory, decode, and execute it. The communication between the processor and the memory is mapped to a channel and synchronized by events, which means that no clock signal need to be explicitly defined. This channel contains methods to request instructions for the memory and to send these instructions back to the processor. Methods are defined by interfaces, which are classes that have virtual objects. Interfaces are also used to define the ports of the modules. For example, when the processor requests a new instruction, it performs the *getNxtWord* method through the port that is connected to the *MemoryCHl* channel. This method is executed by the channel, which contains the PC register to address the memory, requesting the information for the memory. The instruction is read from the memory and sent back to the channel. The channel notifies the processor that the instruction is available through an event and awaits for a new request. Figure 2 illustrates the structure of methods used in the processor-memory communication.

```
#include <systemc.h>
#include "R8MemIf.h"
#include "memR8If.h"
class memoryChl: public sc_channel, public R8MemIf,
public memR8If{
public:
typedef sc_lv<16> word_type;
typedef sc_uint<16> ad_type;
void getNxtWord(word_type *word){
      memPosition = PC;
      addressAvailable.notify();
      wait(dataOk);
      *word=localWord;
      PC++; }
void getAddress(ad_type *address){
      wait(addressAvailable);
      *address=memPosition; }
void sendWord(word_type word){
      localWord=word;
      dataOk.notify(); }
private:
  ad_type   PC, memPosition;
  word_type localWord;
  sc_event  addressAvailable, dataOk; };
```

Figure 2 – Partial TLM description for the R8 processor, detailing the processor-memory communication.

The TL abstraction layer permits to describe the processor functionality, abstracting the precise communication mechanisms. The TL description is simulateable, and results can help the designer e.g. to fit the memory size, the number of registers and operations.

4.2. Register transfer level modeling

Once the TL description validated, the next step is the manual refinement to RTL modeling. The computation modules are the first elements to be manually translated to RTL modeling. At this step, flip-flop descriptions, register size and clock cycle accurate control FSMs are defined. The focus changes from functionality to implementation, since a synthesizable description is required.

Keeping channels and first refining the modules allows incremental validation. When computation modules are validated, the communication protocols may be refined. Incremental refinement, simulation, and validation achieve a synthesizable description. Simulation steps are each time longer than the ones performed in TL modeling, since they are executed at clock cycle accurate levels. Also, the number of components in the design increases at each step (registers, state machines, multiplexers, etc).

The channel refinement employs conventional ports, using signals like standard logic or Boolean. The communication protocol is implemented internally to each channel, and the synchronization is carried out using clock signals. The abstraction of data types cannot be used anymore. In the TL description, the channel describing the communication between the processor and the memory contains the PC register. However, in practice this register is an internal processor register. So, this register is moved from the channel to the processor module.

There are no rules for a smooth refinement process. What is interesting to be done is to focus on the problem using a hierarchical approach. From TLM to RTL for example, it is useful to concentrate in refining modules and then refining communication, as done here.

5. Comparison

To compare SystemC to VHDL design and TL to RTL modeling, three implementation types of the R8 processor case study were sought. Several RTL implementations were conducted, various versions in the SystemC language and a "gold" one in the VHDL language. The third type is TL implementations using SystemC, one version of which was implemented. The implementation comparison relies upon two metrics. The first metric is simulation time. The second metric is the size of the automatically generated hardware. Also, qualitative evaluation of the designs was conducted to assess how easy is to learn and use each language.

5.1. Qualitative evaluation

VHDL is a language developed to allow hardware description. It contains specific features to describe hardware structure and behavior including specific control flow structures and specific data types. Once the VHDL description is done, a tool that interprets the hardware description can be used. As hardware and software usually interact in systems, a hardware-software co-simulation may have to be performed during SoC design. With VHDL this process is possible but it has to be done by means of a tool with explicit support to integrate VHDL simulation with software debuggers through the operating system of a host computer. ModelSim is an example of a tool with such support. Moreover, the description of a top module to perform project validation, usually called a *testbench* in VHDL jargon, has to be described using specific VHDL libraries and commands. Thus, to start working with VHDL, a great effort needs to be done to understand it, if the user does not know the language in advance. Also, the user has to dominate the set of specific tools used to capture and validate VHDL designs.

SystemC is a set of classes that extends C++ libraries. It is not considered a specific language to describe hardware yet. It uses C++ data types and control flow. Furthermore, it allows and employs powerful features like C++ template definitions to easily declare objects and manipulate them. Instead of being interpreted, the code that describes the hardware is compiled into an executable file produced using e.g. GNU gcc/g++ or Microsoft Visual C++ compilers. As a C++ environment is used, the effort to describe a cosimulation is much lower compared to VHDL because the same language is used to describe both hardware and software entities without the need of powerful and or heterogeneous supporting tools to perform cosimulation. Since all the power of C++ can be used for simulation/cosimulation, testbenches can capitalize on the use of C++ libraries to validate both the hardware and the software. C++ is a language available to a potentially much larger community of users than VHDL. Although the effort to learn SystemC seems to be lower, this feature may not justify changing the development approach for those used to employ VHDL until now.

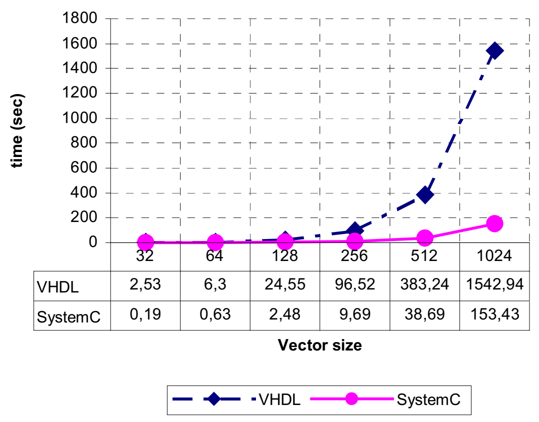
5.2. Simulation time comparison

The simulation time analysis was adopted to compare the efficiency obtained from a hardware design starting from an RTL VHDL description to a TL description in SystemC. The RTL VHDL description is exercised with a clock cycle accurate VHDL simulator, while the SystemC description is compiled code generated by the GNU g++ compiler. VHDL simulation employed Modelsim.

For this process, the same testbench was used with both descriptions, which included running an R8 assembler version of the bubble sort algorithm. Vector sizes are 32, 64, 128, 256, 512 and 1024 16-bit words. Both simulations were run in a Sun Blade 2000, with a 900 MHz single processor and 1Gbyte of main memory.

Figure 3 presents the simulation time in seconds needed to execute the bubble sort algorithm.

As appears in the figure, the SystemC simulation time is nearly an order of magnitude faster than the VHDL simulation. This information is relevant, since, the time taken to simulate large circuits described in VHDL (as found in SoC design) is extremely high and constitutes a main factor in the critical path of complex IC design to fulfill time-to-market schedules. In this way, adoption of SystemC may significantly reduce time taken in design validation, if synthesis can proceed directly from this point to automatic hardware generation procedures.



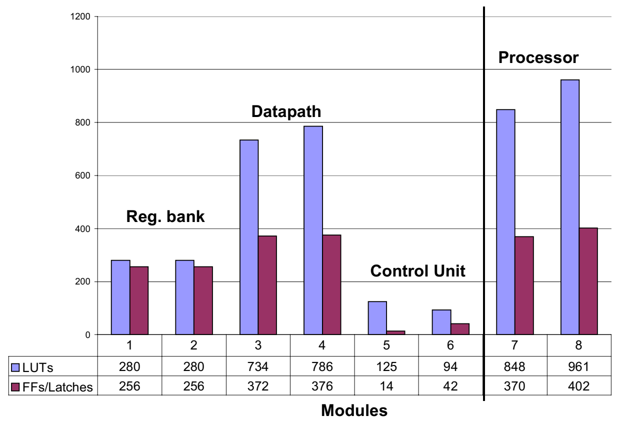
	32	64	128	256	512	1024
VHDL	2,53	6,3	24,55	96,52	383,24	1542,94
SystemC	0,19	0,63	2,48	9,69	38,69	153,43

Figure 3 - RTL versus TL simulation time comparison Time values are available below the graph.

It is important to stress that this case study is about real, working hardware. Both, VHDL and SystemC RTL implementations have been validated by simulation using the same testbench. Both were prototyped in hardware and tested in the same FPGA board, Xess XSV800.

5.3. Hardware size comparison

The next analysis in the context of this case study is the hardware size comparison obtained in the design flow when using either SystemC RTL or VHDL RTL descriptions as input to automatic synthesis. To perform this comparison, an EDIF file was obtained from the SystemC RTL description using the Design Compiler tool from Synopsys. The ISE XST tool imported the EDIF file. The synthesis results obtained from mapping the design to Xilinx Virtex FPGAs appear in Figure 4, comparing the best SystemC RTL version to the VHL RTL version.



	1	2	3	4	5	6	7	8
▨ LUTs	280	280	734	786	125	94	848	961
▨ FFs/Latches	256	256	372	376	14	42	370	402

Figure 4 – Hardware comparison for RTL design of VHDL (odd columns) versus SystemC (even columns). Column pairs correspond to data about register bank (1-2), datapath (3-4), control unit (5-6) and the whole processor (7-8).

Virtex XCV800PQ240 was the target FPGA. The analyzed size metric was the amount of LUTs/FFs used by each hardware description. The size obtained for each module is similar for both, VHDL and SystemC. The last four bars show the total hardware size achieved by both synthesis processes. Processor data is not equal to the sum of datapath and control unit data, due to overall synthesis optimizations. The differences are within 15% for LUTs and within 10% for memory elements. Equivalent gate count estimation furnished by the XST tool is 9,032 for the VHDL version and 9,682 for the SystemC version.

6. Conclusions and future work

The results presented in this work stress the facility to describe hardware using SystemC, which provides a faster simulation time, when compared to VHDL. The gain in simulation time approaches an order of magnitude. Also, the hardware size obtained from the RTL SystemC description synthesis is comparable to that obtained from the synthesis of an equivalent RTL VHDL description. However this was not a straightforward process. Much experimentation with SystemC coding styles was done to achieve good synthesis results, indicating a certain level of immaturity of SystemC RTL synthesis tools.

SystemC stands as an interesting language option to describe hardware at high abstraction levels, because it smoothes the design flow from higher abstraction levels to lower abstraction levels. SystemC is efficient to validate hardware when comparing to VHDL, and potentially enables a much wider community to design hardware.

It is perfectly reasonable to argue that the results reported here are preliminary. Nonetheless, early results in an ongoing work with bigger examples have revealed that the SystemC approach to synthesis has area-scaling characteristics similar to that of VHDL. Although to date no tool is available to automatically translate TL models to hardware, these differ little enough from RTL models to allow considering that automatic synthesis is not a long way from real life design flows.

Even though the size of the reported case study is not at all that of a real SoC, the R8 processor design comprises all major hardware module types expected to be present in real SoCs. This includes state machines (the control unit of the processor) arithmetic logic (the ALU), multiport memories (the register bank) and wide interconnection resources (the ALU buses and the external memory interface). Such reasoning turns the case study choice adequate to investigate SoC design techniques.

Ongoing and future works include describing NoCs inside SoCs at the TL abstraction level in SystemC. The RTL case studies reported here have been prototyped in hardware. From several SystemC RTL versions that simulated identically to the VHDL RTL hardware the one that we managed to run in hardware up to now is significantly bigger than the version reported here (12,081 equivalent gates). We are investigating this discrepancy. Other relevant implementation comparison data includes the quality of the design with regard to maximum operating clock frequency for both, FPGAs and ASICs.

7. References

[1] Martin, G.; Chang, H. **Tutorial – System on Chip Design**. In: ISIC´01, Singapore, 2001.

[2] Semiconductor Industry Association. **International Technology Roadmap for Semiconductors – Executive Summary**, 2001. Available at http://public.itrs.net.

[3] Gupta R.K.; et al. **Introducing Core-Based System Design**. IEEE Design & Test of Computers, 14(**4**), Oct.-Dez. 1997. pp. 15-25.

[4] Bergamaschi, R. A.; et al. **Automating the design of SoCs using cores.** IEEE Design & Test of Computers, 18(**5**), Sept.-Oct. 2001. pp. 32-45.

[5] Dally, W.J.; Towles, B. **Route packets, not wires: on-chip interconnection networks**. In: DAC´01, 2001. pp. 684-689.

[6] Benini L.; et. al. **Networks on chips: a new SoC paradigm**. IEEE Computer, 35(**1**), Jan. 2002. pp. 70-78.

[7] Virtual Socket Interface Alliance. **VSI Alliance Architecture Document**. Version 1.0. 1997. 69 pages. Available at http://www.vsi.org/

[8] OCP International Partnership. **Open Core Protocol Specification**. Release 1.0. 2001. 202 pages. Available at http://www.ocpip.org/socket/ocpspec/

[9] Hartenstein, R. **A decade of reconfigurable computing: a visionary retrospective**. Tutorial, In: DATE´01. 2001. 9 p.

[10] Keutzer, K; et. Al. **System-level design: orthogonalization of concerns and platform-based design.** IEEE Transactions on Computer-Aided Design, 19(**12**), Dec. 2000. pp. 1523-1543.

[11] De Micheli, G.; Gupta, R. **Hardware/software co-design**. Proceedings of the IEEE, 85(3), March, 1997. pp 349-365.

[12] Jacome, M.; Peixoto, H. **A survey of digital design reuse**. IEEE Design and Test of Computers, May-June, 2001.

[13] Bailey, B.; Gajski, D. **RTL semantics and methodology**. In: ISSS´01, Montréal, 2001. pp. 69-74.

[14] Pasricha, S. **Transaction level modeling of SoC with SystemC 2.0**. In: Synopsys Users Group Conference - SNUG´2002. India, 2002.

[15] Arnout, G. **EDA moving up, again!** In: System-Level Design: Here And Now. OSCI technology symposium at DAC´02, 2002. Available at http://www.systemc.org/projects/sitedocs/document/DAC_2002

[16] Haverinen, A. et al. **SystemC based SoC communication modeling for the OCP protocol**. White Paper. V1.0, October, 2002. 39 pages. Available at http://www.ocpip.org/data/systemc.pdf

[17] Synopsys Inc. **SystemC Version 2.0 Users Guide**. 2003. Available at: www.systemc.org

[18] Thorsten, G. et al. **System Design with SystemC**. Kluwer Academic Publishers, Boston, 217 pages, 2002.

[19] Moraes, F. G., and Calazans, N. L. V. **R8 Processor Specification and Design Guidelines**. 2003. Available at: http://www.inf.pucrs.br/~gaph