Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0

M. Caldari[*], *M. Conti*[*], *M. Coppola*[**], *S. Curaba*[**], *L. Pieralisi*[*], *C. Turchetti*[*]

[*] University of Ancona, via Brecce Bianche, I-60131, Ancona, Italy

[**] STMicroelectronics, Grenoble, France

Abstract

The concept of a SOC platform architecture introduces the concept of a communication infrastructure. In the transaction-level a finite set of architecture components (memories, arithmetic units, address generators, caches, etc) communicate among each other over shared resources (buses). Until recently, modeling architectures required pin-level hardware descriptions, typically coded in RTL. Great effort is required to design and verify the models, and simulation at this level of detail is tediously slow. Transaction level modeling is the solution. Transaction level models (TLMs) effectively create an executable platform model that simulates orders of magnitude faster than a RTL model.

In this paper, we present a SystemC 2.0 TLM of the AMBA architecture developed by ARM, oriented to SOC platform architectures.

1. Introduction

Evidently, system-on-chip era is creating many new challenges to the current design flow. Increased demand for complexity captures and consistency in hardware modeling, especially for SOC design, has led to the development of new modeling methodologies and corresponding simulation engines. To specify, design, and implement such complex systems, incorporating functionality implemented in both hardware and software forms, we are compelled to move on from HDLs of old. We must also move beyond the RT level of abstraction used with these HDLs. We need to move to what has been termed the "system level" of design with a modeling language that can support this level. Several modeling methodologies have been proposed in the past years for increasing the level of abstraction and enabling hardware-software co-design [6,8,9]. Specification at higher levels of abstraction is possible in environments such as SystemC 2.0 [8,10]. SystemC 2.0 is an emerging standard modeling platform based on C++ that supports design abstraction at the RTL, behavioral and system level.

Apart from the modeling benefits of C++ [2] such as data abstraction, modularity, and object orientation, advantages of SystemC 2.0 include the establishment of a common design environment consisting of C++ libraries, models and tools providing the ability to exchange and reuse IP easily and efficiently across different levels of abstraction. In this paper, we show how the communication classes available in SystemC 2.0 can be used in order to produce very fast transaction-level bus models suitable for SOC platform architectures. The key in efficient bus modeling is to create code in a way that allows simulation to run very fast. The only way to achieve this goal is to write code completely detached from hardware block implementations, raising the abstraction level [9] and opening a new scenario in model development. In particular, we created a SystemC 2.0 Bus-cycle-accurate (BCA) model of AMBA specification developed by Arm. The model that we wrote supports the full AMBA rev2.0 specification and the Arm Multi-layer AHB. The remainder of this paper is organized as follows. In Section 2, we explain the transaction level modeling style. In Section 3, we describe the AMBA model, with C++ class descriptions and implementation methodology. In Section 4, we show the test environments. In Section 5, we report the performance evaluation and test results. Finally, Section 6 draws the conclusion.

2. Transaction-level modeling

SystemC 2.0 introduces a new set of features for generalized modeling of communication and synchronization [9,10]. These are: channels, interfaces and events. An interface defines a set of methods, but does not implement these methods. It is a pure functional object without any data in order not to anticipate implementation details. A channel implements one or more interfaces. A port enables a module and hence its processes, accessing a channel's interface. A port is defined in terms of an interface type, which means that the port can be used only with channels implementing that interface type. With channels, there is a distinction between so-called *primitive channels* and *hierarchical channels*.

Primitive channels do not exhibit any visible structure, do not contain processes, and cannot directly access other primitive channels. Hierarchical channels, on the other hand, are modules, which means they can have structure, they can contain other modules and processes, and they can directly access other channels. The use of interfaces enables a very powerful scheme called *interface-method-call* (IMC). IMC refers to a process calling an interface method of a channel. The interface method is implemented in the channel, but it is executed in the context of the process. At Transaction-Level, communication mechanisms such as buses or FIFOs are modeled as channels, and are presented to modules using SystemC 2.0 interface classes. Transaction requests take place by calling interface functions of these channels models, which encapsulate low-level details of the information exchange. In other words, at the transaction-level, the emphasis is more on the functionality of the data transfers-what data are transferred to and from what locations- and less on their actual implementation (that is, on the actual protocol used for data transfer). In transaction level modeling, synchronization details are typically abstracted into the categories of blocking and non-blocking I/O, and in the case of buses, priorities may be assigned to clients, and arbitration can be modeled in a centralized way. Transaction-level modeling also enables higher simulation speed than pin-based interfaces [3], through the suppression of "uninteresting" details [7]. For example, in the real world a large burst-mode transfer may take many actual clock cycles to complete. In most of these clock cycles, the bus is merely doing routine work and those clients that have pending bus requests are just waiting. If we view the burst-mode transfer as a single operation, there is no need to devote simulation time to these "uninteresting" clock cycles. Depending on whether the model needs to be bus-cycle-accurate (BCA) or not, different strategies can be applied to take advantage of this, resulting in significant savings in simulation time. As we will demonstrate in the next sections, even when a transactional-level model needs to be cycle accurate, it still may simulate much faster than a typical cycle-accurate RTL model.

3. AMBA model overview

The AMBA specification defines an on-chip communication standard for designing high performance embedded micro controllers [1]. Three different bus specifications are defined within AMBA architecture:

- the advanced high-performance bus AHB;
- the advanced peripheral bus APB;
- the advanced system bus ASB.

Our goal was to create cycle-accurate TLMs for the AHB and the APB buses. This would allow effective incorporation of buses into SOC modeling platforms, with appropriate communication interfaces and correct timing. Moreover, we had wanted to build models that execute faster than others in usual simulation environments. Our AMBA model shows how to obtain a clock-accurate simulation without using RTL [7] specific hardware signals and components; that is to say, we developed a model with a high-level of abstraction [11] that does not need to describe all hardware details that the real architecture needs.

The model, as we will explain later, uses the dynamic sensitivity implemented by SystemC 2.0 in order to avoid useless function calls when it is not useful, at simulation level, holding the model running. Being a transaction-level model, it can be used to simulate the AHB and APB bus protocols in a correct way, keeping the right control options, but masking them in a layer whose implementation is completely hidden to the user. This is very important, since the composition of a high level behavioral model for an embedded system must be based [4] on protocol refinement. AMBA TLMs are built using all necessary building blocks for modeling standard channels available in SystemC 2.0, that is, interfaces, ports and channels.

3.1 Classes' structure

In this Section, we describe the C++ classes' structure to build on-chip bus models. The classes' structure is based on **SC_interface**, **SC_channel** (SC_Module) and **SC_port** (SystemC 2.0) classes but it does not use **SC_primary_channel**. In Fig. 1, we show the APB classes' structure expressed using OMT notation. Notice that for the AHB we kept the same structure.

The vertical line with a triangle denotes class inheritance. An arrowhead line is used to represent aggregate dependency between classes, i.e. one class is composed in part from another class. This aggregation can be further redefined. Reference aggregation, graphically denoted as a black rhombus, means the whole object maintains a pointer or a reference to its part, while value aggregation, graphically denoted as a white rhombus, means the whole object is included. Starting from the high level, we declared a bus like a template class that uses template arguments as the ones used for the related interface. The user defines such arguments, and adds further attributes that can be used by higher-level communications, in order to implement custom transfer over the physical AMBA protocol. Notice that the programmer cannot use a completely user-defined class for the attributes, because at least the controls of protocol must be declared inside it (with a particular attention to *hready* signal, as we will explain later, when we will describe this important feature of our model).

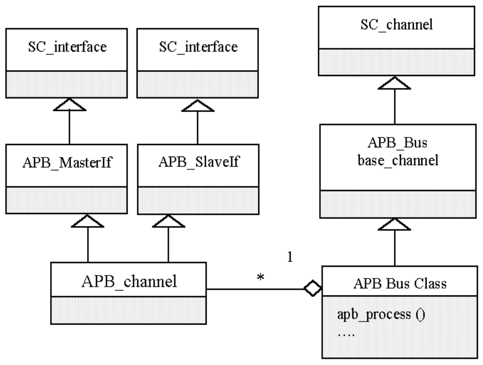

Figure 1: AMBA model classes' structure

The Bus class (see Fig. 1) is at the top level of the hierarchy and is used as interface with the ports that connect the user's test bench with the Bus Class.

The Bus class provides all the constructs that make the communication between behaviors possible. In particular, we use dynamic instantiation that creates a new class **X_channel** (where **X** represents bus protocol) for every master or slave port instantiated into the model. That is, we create a Bus as an object that contains several channels (**X_channel** is not a **SC_channel**, it is a normal C++ object), just to respect the modularity of an object oriented reuse specification. The Bus class inherits the class Bus Base Channel (see Fig. 1), where we perform several activities, first of all a mechanism of data transfer totally hidden to the user, which is used just to simulate the data transfer in a clock-accurate manner. Finally, the class Bus base channel inherits the (SystemC 2.0) **SC_channel** that is a base class for all the SystemC 2.0 hierarchical channels, and is where the data transfer is really performed. The Bus Class is used also to declare the **X_process()** function (inside Bus class, where **X** represents bus protocol), implemented as a SC_method.

The **X_process()** method, that is, the bus core, performs the bus operations and manages the complete protocol. For the AMBA model development, we used the two-phase synchronization scheme, so, the **X_process()** method is sensible to negative edge of the clock. This because certain modules (masters and slaves) are active on the rising edge of the clock, while other modules (bus) are active on the falling edge. Because the Bus now executes on the falling edge of the clock, we can be sure that by the time the bus executes it has gathered all of the requests for this bus cycle, since all masters execute on the rising edge, assuring deterministic design.

3.2 A state-oriented model

A state-oriented model is one that represents the system as a set of states and a set of transitions between them, which are triggered by external events. A finite-state machine (FSM) is an example of a state-oriented model.

Basically, the FSM model consists of a set of states, a set of transitions between states, and a set of actions associated with these states or transitions.

In our model, we concentrated exclusively in the **program-state machine** (PSM)[5], that is, an instance of a heterogeneous model that integrates a **hierarchical concurrent finite-state-machine** (HCFSM) with a programming language paradigm. The HCFSM is essentially an extension of the FSM model, which adds support for hierarchy and concurrency, thus eliminating the potential for state and arc explosion that occurred when describing hierarchical arc concurrent system with FSM models. Like the FSM, the HCFSM model consists of a set of states and a set of transitions. Unlike the FSM, however, in the HCFSM each state can be further decomposed into concurrent sub states, which execute in parallel and communicate through global variables. As already say, we focused on the PSM model.

This model consists of a hierarchy of program-states, in which each program state represents a distinct mode of computation [5]. At any given time, only a subset of program-states will be active, that is, actively carrying out their computation. Both the APB and AHB **X_process()** methods have been implemented using PSM model, as we are going to explain.

3.3 The APB bus core

The **X_process()** method present in every bus class, is, as already described in Section 3.1, the real core of the bus model. We chose for the **apb_process()** implementation the Program State Machine model (see section 3.2).

We can explain the PSM implementation of the APB Bus **apb_process()** method as follows. The APB IDLE state mapped in our model [1] is just used to check the opcode *pwrite* signal and the *pselx* signal (selected slave port), so the transaction continues taking into consideration these initial conditions. Hereinafter we show the pseudo-code of PSM in the **apb_process()** method (IDLE state):

```
template <class MasterAttr,class SlaveAttr>
void ApbBus<MasterAttr,SlaveAttr>::apb_process()
{
 ...
 case IDLE:
  if (opctmp) {
              current=TX_WRITE;
              //(Sequence of operations)
              return;
              }
  else {
        current=TX_READ;
        //(Sequence of operations)
        return;
       }
 return;
}
```

Every single state performs different operations, depending on the past state and the future state. In this way, we manage the Bus protocol in every clock cycle **without** having to implement any system that describes protocol signals (e.g. the *penable* signal does not exist in our model that remains clock accurate). This avoids an enormous computation burden, so simulation is very fast as shown in our results. Using a PSM, it is quite simple to simulate a given communication protocol without the concept, costly in terms of performance, of hardware signal (SystemC 2.0 RTL style), reaching a higher level of abstraction and consequently a faster simulation.

Moreover, there is a further option that permits the simulation to run faster. Our system is composed by different components (System C 2.0 modules) that run in parallel each performing its task. But when a Module (e.g. a Bus module) during a determined period of time has nothing to do, it is useless and costly in terms of CPU time to continue to call it by the system scheduler, that is, the class that manages the simulation. Thus, we implemented the so-called "*de-scheduling*" function by dynamic sensitivity, that avoids useless calls to the **apb_process()** method when it has nothing to perform and permits to "*re-schedule*" (notify) it when a determined event comes. With the bus control in the form of a PSM, it was easy to manage the *de-scheduling* option since in every state we have enough information that permit us to know the bus status, and the need of computation.

In the APB Bus we adopted the choice of to "*de-schedule* " the bus when the Master Interface is not ready to send the data; that is, the bus class is waiting for an event (**next_trigger (**event**)**) coming from master's module. When the master starts communication, in the same time it notifies the event associated to the bus, so, static sensibility of the bus (negative edge of the clock) re-becomes active. We would like to note that if we had used a hardware-like implementation, correct use of the *de-scheduling* option, if accomplishable, would have been more tedious and difficult.

3.4 The AHB bus core

We have already described the PSM model of **X_process()** method in a bus class with the APB example. We now consider implementation of the **ahb_process()** method within AHB Bus. We described the Bus as a PSM where each state represents in the same time the transfer that we have just performed and the transfer that we have to perform in the current cycle. We made this choice taking into consideration the pipelined nature of AHB Bus that implies the transfer of the current control signals and at the same time the transfer of the data referring to the previous cycle.

In this way, the **ahb_process()** method can be considered as a table where we describe all the possible transactions that the AHB Bus can perform.

Looking at this table in a fixed clock-cycle we know exactly what we must do and who must do it, e.g. a read transfer to slave port 3. The user must only supply the correct control signals, respecting the protocol implementation, but it is important to notice that at bus level we do not simulate the hardware RTL signals transfer to keep the correct timing. Instead, we use the blocking methods just to set several classes that manage the addresses and compute the right state for the PSM, that must only perform the code relative at the actual state, resulting in a very low computational burden. We would notify that an effort was made to manage the PSM when a bus handover occurs, because with the pipelined nature of the data and address bus we must continue the data transfer of the old master while charging the new control signals for the new master.

This results in different PSM states to be written on purpose for the bus handover. We would like to notice that our model is quite detached from the hardware level. In fact, just as example, the classes that we used to describe the AHB Bus do not reflect any hardware block that anyone can find in the Arm AMBA specification. We have just described the AHB like a so-called black box within which we mask the hardware implementation with a high-level abstract implementation.

Obviously, the de-scheduling problem in a bus like AHB holds a lot of interest, because in a so complex object we looked a lot for a method that avoids useless function calls, improving the simulation speed.

Our choice was de-scheduling (or masking) the bus work with respect to *hready* signal, the signal used by the slave ports to manage a correct transfer, by dynamic sensitivity of **ahb_process()** method. By the AMBA protocol [1] we know (see Fig. 2) that when a slave port sets *hready* low and *hresp* equal to *okay*, the transfer is in a so-called "sleepy state".

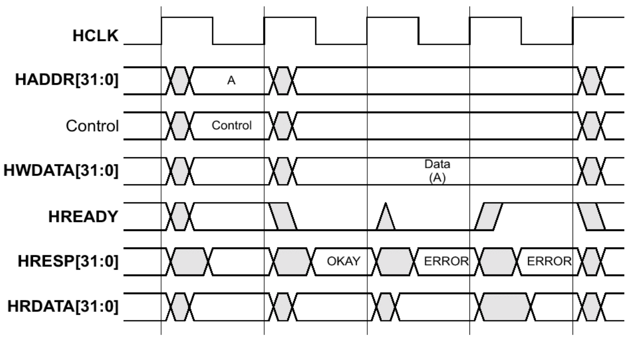

Figure 2: HREADY signal example

That is to say, the bus has nothing to perform in a determined clock cycle. If we do not use a de-scheduling option the scheduler every cycle would call the bus **ahb_process()** method (Bus core) waiting on a response different from *okay* or a *hready= high*, resulting in a considerable performance reduction.

We resolved the de-scheduling problem by mapping the *hready* and *hresp* signals (declared as Boolean and enumerated values) as an event: the event that has the capability to supersede static sensitivity in **ahb_process()** method when no transfer is needed and that in the same time can reschedule (*hready* rises or *hresp* differs from *okay*) the bus when a transfer must be performed.

In this elegant way, the Bus class avoids useless routine work during the time *hready* is held low, lowing computational burden and improving simulation speed.

4. Test environment

The following section shows the test bench environment that we chose to test the AMBA Model. We used CThread objects, in order to simulate every Master and Slave port, triggered by the positive edge of the clock. A CThread consists of a function that is executed in an infinite loop, obviously with blocking conditions inside (conditions that are waiting a time-dependent event to restart the block of code, typical transactions at TL). We chose two different types of test bench for the APB Bus and the AHB Bus. In the APB Bus we chose a single CThread in both sides (Master, Slave), because the protocol is quite simple and for a correct simulation we do not need to split write and read operations in two separate CThreads. On the contrary, with AHB specification, we chose to split read and write operations in Slave Side with two CThreads (Fig. 3). In the AHB Master side we used also two CThread processes (Fig. 3), the first performing all the transfers (Main), the second used just to receive the data coming from slave ports during read transfer.

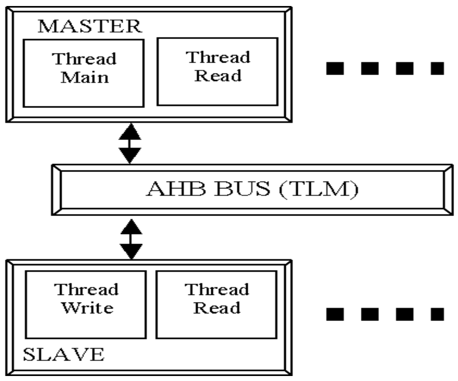

Figure 3: AHB testbench configuration

5. Performance

The following section describes improvement on the simulation speed of the AMBA model. Before it was implemented, we examined the SystemC 2.0 TLM of STBus, ST proprietary standard on-chip bus. It was a static C model providing clock cycle-accurate simulation. Next, it has been integrated by ST designers in SystemC 2.0 environment directly at TL.

STBus TLM simulated slow due to an extremely accurate hardware modeling concerning all signals (modeled by variables, not by SystemC 2.0 signals) and blocks, closer to a RTL model than a TLM. Each hardware block had its class representation, with all output signals and input signals, e.g. the class arbiter reflects perfectly the arbiter hardware block, and each block was simulated cycle per cycle (STBus SystemC 2.0 TLM does not implement dynamic sensitivity). Every effort made in order to construct an abstract simulation engine, quite detached from HDLs was wasted in a communication channel that simulated the exact hardware structure. We realized that this was not the correct way to operate, and thus we examined methods to define a bus implementation in a more abstract way based on PSM and de-scheduling (dynamic sensitivity) concepts as already described. Our implementation does not reflect any hardware block in AMBA specification; that is, we wrote a model that executes in the same manner as hardware but it does not use any hardware concept. At the end of code writing, we were faced with the evaluation and test of our TLMs. In Fig. 4, we report the results of several comparisons between SystemC 2.0 TLMs of STBus, APBBus, AHBBus. We ran several simulations using two kinds of bus traffic and two types of Sun workstation: Ultra 60 and Blade 1000.

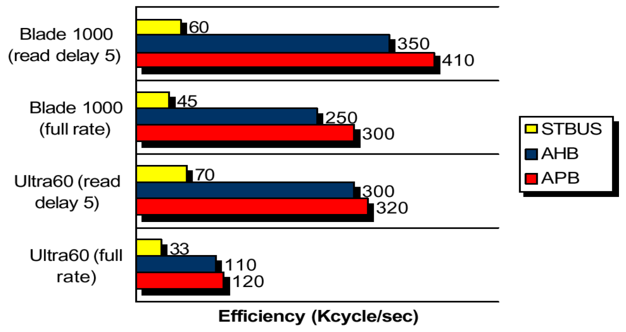

Figure 4: Simulation efficiency for various architectures and model delay options

The first test simulates full rate traffic with alternative write and read transfers; that is, the bus performs a transfer every cycle. The second simulates the same transfers, but with a read cycle performed with five cycles delay. Simulation results were as expected. The models of APB and AHB Bus run at same speed in both cases, whereas the model of STBus is much slower. This shows that new design methodologies achieved our goal.

The second comparison is between two models of AHB Bus written at different levels of abstraction, our SystemC 2.0 TLM and a proprietary model used by ST designers written in SystemC 2.0 at RTL.

Both models have been tested with same traffic architecture: a DMA controller alternatively drives write and read transfers to RAM memory (see Fig. 5).

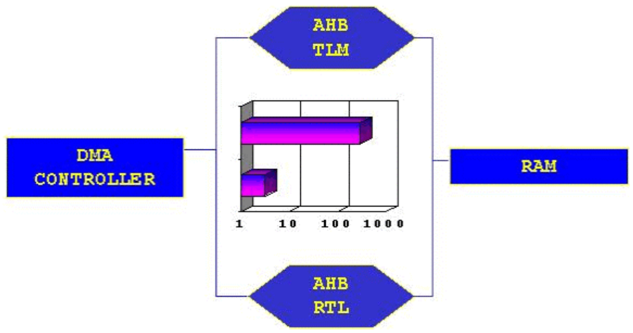

Figure 5: Testbench used to test performance

The following model assumptions are made. The delay time for a write transfer is 0, while the delay for a read transfer is one clock cycle; that is to say, the slave module holds low the signal *hready* with *hresp=okay* during the first read cycle. The DMA needs four clock cycles in order to pass to next write and read operations. There is no delay state between write and read operations. The simulation has run on Ultra 60 machine. The results that we obtained were as expected: both methodologies that we implemented (PSM and dynamic sensitivity) gained a lot in comparison to a hardware level design (Tab. 1).

	AHB SystemC 2.0 RTL Model	AHB SystemC 2.0 TLM
Kcycles/sec	3	300

Table 1: Comparison between SystemC 2.0 TLM and RTL model (Sun Ultra 60)

Moreover, the choice of de-scheduling becomes effective when in the bus there is merely routine work, yielding faster simulation since RTL model continues to perform operations in these clock cycles. In Table 1, we report the results that give a correct and objective dimension of the gain-factor magnitude. We must also keep in mind that a TLM is also more effective for IP reuse [4], so, with an eye to the future, it will gain more avoiding the need of test bench rewriting.

6. Conclusion

The TLMs of AMBA Architecture provide some important results for SOC modeling. First, we showed that with a higher level of abstraction than RTL, we gain two orders of magnitude in simulation speed. Assuming the same simulation is performed on the same machine, if SystemC 2.0 TLM requires a day to run, the SystemC 2.0 RTL model requires almost one hundred days. This very important result removes any residual doubt in TL modeling effectiveness. Second, in a bus implementation, the de-scheduling feature, implemented by dynamic sensitivity, allows the simulation to run faster, avoiding useless function calls, showing that it can become a must in bus modeling.

Finally, the PSM implementation opens a new scenario in bus modeling, providing the user with a robust method in order to create models in a simple way, avoiding useless computational burden. In this way, several other buses can be developed keeping the idea of a single central unit that manages communication in a faster manner, especially without waste of precious CPU time.

Further works are in progress in order to integrate the AMBA TLMs within a SOC platform, oriented to power estimation at system level. We expect that these works will produce further improvements in the modeling concept and implementation.

Acknowledgments

This research has been sponsored in part by EU Medea+.

References

[1] AMBA Specification (rev2.0) and Multi layer AHB specification, Arm: http://www.arm.com, 2001.

[2] M. Caldari, M. Conti, M. Coppola, M. Giuliodori, C. Turchetti: *"C++ based System-on-chip Design"* IEEE Canadian Journal of Electrical and computer Engineering, vol. 26, no. 3/4, July/Oct. 2001, pp. 115-123.

[3] CoCentric System Studio Data Sheet, Synopsys http://www.synopsys.com, 2002.

[4] R. Domer, Daniel D. Gajski: "*Reuse and protection of Intellectual Property in the SpecC system*", University of California, Irvine, http://www.ics.uci.edu.

[5] Daniel D.Gajski, Jianwen Zhu, Rainer Domer *"Essential issues in co-design"* University of California, Irvine Technical report June 1997, http://www.ics.uci.edu.

[6] J. Gerlach, W. Rosenstiel *"System level design using SystemC modeling platform "* University of Tubingen, Germany, www-ti.informatik.uni-tuebingen.de/~systemc.

[7] A.Gerstlauer, S.Zhao, D.Gajski, A.Horak: *"SpecC System-level design methodology applied to design of a GSM Vocoder"* University of California, Irvine and Motorola Semiconductor products sector, http://www.ics.uci.edu.

[8] T. Grotker, S. Liao, G. Martin, S. Swan: *"System design with SystemC"* Kluwer Academic Publishers, 2002.

[9] Preeti Ranjan Panda: "SystemC – A modeling platform supporting multiple design abstractions", Synopsys Inc, http://www.synopsys.com.

[10] Open SystemC Iniative (OSCI), SystemC documentation: http://www.systemc.org, 2001.

[11] Kjetil Svarstad, Gabriela Nicolescu, Ahmed A. Jerraya: *"A model for Describing Communication between Aggregate Objects in the Specification and Design of Embedded systems"* SINTEF Telecom and Informatics, TIMA Laboratory, SLS group, http://www.systemc.org.