The Role of Model-Level Transactors and UML in Functional Prototyping of Systems-on-Chip: A Software-Radio Application

Alexandre Chureau, Yvon Savaria, El Mostapha Aboulhamid

To cite this version:

Alexandre Chureau, Yvon Savaria, El Mostapha Aboulhamid. The Role of Model-Level Transactors and UML in Functional Prototyping of Systems-on-Chip: A Software-Radio Application. DATE'05, Mar 2005, Munich, Germany. pp.698-703. hal-00181191

HAL Id: hal-00181191
https://hal.science/hal-00181191

Submitted on 23 Oct 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Role of Model-Level Transactors and UML in Functional Prototyping of Systems-on-Chip: a Software-radio Application

Alexandre Chureau[1] Yvon Savaria[1] El Mostapha Aboulhamid[2]

[1]École Polytechnique de Montréal, Dép. de Génie Électrique
C.P. 6079, succ. centre-ville, Montréal, Canada, {chureau|savaria@grm.polymtl.ca}
[2]Université de Montréal, Dép. d'Informatique et de Recherche Opérationnelle
C.P. 6128, succ. centre-ville, Montréal, Canada, em.aboulhamid@umontreal.ca

Abstract

Developing a functional prototype of a system-on-chip provides a unifying vehicle for model validation and system refinement. Keeping the prototype executable across several abstraction levels, clock domains and design tools is a key requirement to effective prototyping. This paper presents how model-level transactors address design heterogeneity by unifying event-based and cycle-based worlds from specification to implementation. Transactors are used to build a functional prototype of a software-radio component. An executable UML model is bridged to a hardware abstraction of a radio stream developed with Simulink to implement a realistic and working prototype. Model validation and performance measurements are realized through prototype execution and real-time monitoring.

1. Introduction

The development of complex systems on chip (SoC) faces a paradox. Developing a system of a reasonable scale requires an extensive analysis and design phase before implementation. But the very nature of these systems is dynamic and intricate, thus requiring some sort of implementation to carry out a knowledgeable and effective design. Hence, it has been accepted that design and implementation are two activities that are profitably carried out at once.

Leveraging this association between design and implementation, the concept of functional prototype has emerged as a means to handle the rising complexity of design. A functional prototype is a machine-executable specification of the system and its operating environment that is used for early system validation. More generally, a prototype is a primitive but working version of the system that can be executed and observed in real-time using real world stimuli.

In electronic design, the prototyping activity ranges from a software specification of the final product to an implementation on a FPGA-based prototyping board. What differentiates a prototype from a simulated model is the ability of the former to work in real-time, maybe at some low but useful performance, and to allow different blocks to execute seamlessly while the system appears as a coherent whole.

The concept has been around since top down methodologies took over ASIC design. Early prototypes have been used for formal specifications and model validation [1, 2]. The functional virtual prototype recently presented by Cadence is at the heart of its proprietary verification methodology [3]. This virtual prototype is used as a unifying vehicle expressed at the transaction abstraction level. So far, functional prototyping has been hardware-centric, and meant to represent hardware systems at high level of abstraction. This is becoming inefficient and unnatural, as software plays an increasing role in systems-on-chip. In this paper we make extensive use of the software-centric UML-RT (Unified Modeling Language – Real-Time) to build a more neutral, yet executable, functional prototype of a SoC.

This paper presents a transaction mechanism that enables several levels of abstraction to be represented in a functional prototype built around UML. This leads to:

- Matching the expressiveness of a language with the desired level of details across the prototype.
- Distributing the development effort among different experts working with different tools.
- Providing an executable and unifying specification built in a well accepted modeling language, UML.

Model level transactors are used to couple asynchronous components such as behavioral description with synchronous components. Multiple abstraction levels coexist and collaborate through a unifying software backplane described in UML-RT. The resulting prototype is executable, observable and compatible with hardware descriptions and verification environments.

The first section of the paper describes the functional prototype and the role of UML-RT in its development. The next section presents the details of model transactors

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

and their exact role in the prototype. The final section presents how a complete functional prototype was built for a software-defined radio component. Some observations and recommendations conclude the paper.

2. Prototyping and the Real World

2.1. Concept

The functional prototype of an electronic system is an unambiguous executable specification of the system coupled with a realistic test environment. It is built from the paper specification in collaboration with design and verification engineers, as well as application specialists. The executable specification is a model that iterative refinement will transform into a network of collaborating objects, much like the final system. The test environment provides realistic data right from the beginning of the design cycle to allow model observation and validation. It can be put together in the most efficient manner with fourth generation languages and verification IPs. If the proper tools are used, the model will gradually become the system, and the test environment will be reused all along the development, increasing design reuse and productivity [4, 5].

An important aspect of the functional prototype has been purposely left out of this discussion so far: the communication mechanisms between the different parts of the prototype. These mechanisms play a crucial role in preserving the executability of a prototype. One mechanism is necessary to transfer test data between the test environment and the executable model. Another similar mechanism is used by the model to communicate with lower-abstraction components introduced during the refinement, e.g. hard IP cores. The next two sections will describe these important mechanisms in more detail, while the rest of the paper is devoted to their implementation and analysis.

2.2. Connecting the World Around

The primary role of the test infrastructure in functional prototyping is to act as the world around the system being developed. The test infrastructure has the following characteristics:

1. It must provide realistic data to the executable model.
2. It is described at high-level of abstraction for speed of development and convenience.
3. It improves controllability and observability over the test scenarios.
4. In some cases, the test infrastructure must act as a placeholder for the analog circuitry that will convey the signal to the system.

It is in the context of an executable specification that the importance of testing arises so early: besides the extensive verification that will be performed along the design cycle, test is used early to validate the model. The executable model will respond to stimuli, encouraging test reuse from the executable specification down to implementation.

The connection mechanisms between the test environment and the model should be minimally intrusive in order to stay clear of the model execution and refinement. Extensive interaction between the test and the model might be necessary to feed a model that is dataflow dominated. To allow large amounts of data to be processed, the communication mechanisms should use fast transfer techniques such as shared memory, memory mapped files and sockets.

2.3. Connecting the World Below

As model refinement progresses, several clocked components expressed at lower levels of abstraction or described in different models of computations may get involved in model execution. Some hardware components may be used to enhance performance of critical parts or to reuse some existing hardware IP. Eventually, the executable model will reach implementation, meeting all temporal constraints of the application.

Although this is a powerful property of functional prototyping, model heterogeneity raises a communication issue: most hardware functions are synchronized on a clock signal, while the initial model behavior is described in UML using asynchronous operation calls, which leads to some sort of globally asynchronous locally synchronous system. In all cases, preserving the model integrity and executability after the inclusion of these clocked components is a priority. This is part of the challenge of seamlessly connecting the executable model to the world below with model level transactors.

Somewhat similar concepts are found in the Ptolemy project, where a framework enables collaboration of executable models built under different models of computations [12]. The framework uses domain-polymorphic actors to wrap executable entities and has its own design entry interface. In contrast, model-level transactors live at the boundary of real-time and simulated-time domains, which encompasses model-of-computation boundaries. In that sense, model-level transactors are closer to existing design and implementation tools.

The MASCOT methodology [13] implements a communication protocol between control and dataflow domains, defined in SDL and Matlab respectively. This co-simulation method is also bound to models of computations, which is an issue efficiently addressed by tool vendors. Mathworks has since launched an integrated tool that manages state machine and dataflow models transparently. The primary issue addressed by model level transactors is how components developed in different time domains (real vs. simulated) can cooperate to form a complete functional prototype. The specificity of tools and vendor methodologies prevent this kind of interaction.

Figure 1 puts in context the executable model, the test infrastructure and a hardware component model. The stopwatch emphasizes the fact that the model is executed in real-time, not simulated.

IEEE COMPUTER SOCIETY

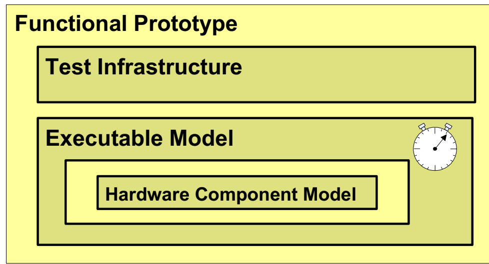

Figure 1. Overview of the functional prototype

3. Event-Based UML and SoC prototyping

3.1. UML for the Executable Model

The Unified Modeling Language (UML) offers all the objects and diagrams needed to build an unambiguous specification of a SoC. Originally a software modeling tool, UML has recently been extended to model real-time systems, an extension we will refer to as UML-RT [6]. Its usage in SoC design is an active field of research [11]. The usage of UML-RT allows building a model of a SoC that is not biased towards a hardware or software implementation. The basic architectural unit is a capsule and its associated state machine, which describe the structure and the behavior of a system respectively.

A capsule is a stereotype of a class that is the equivalent of a VHDL entity, a SystemC SC_MODULE or an independent software execution thread. Some vendors of UML editing tools have implemented efficient model to code translation, which will be used in this paper [8]. Populating the capsule's state machine with platform-independent C++ code produces a complete executable model of the system under development, which can be executed and profiled on different CPUs. In that regard, this approach differs from the Seamless co-verification environment [14], which runs software over an instruction-set-simulator (ISS) of the target CPU.

As introduced earlier, this executable model is the entity that must be connected to the test infrastructure as well as to the hardware implementation components used during the refinement. In both cases, the communication mechanism must resolve real-time events vs. cycle-based events, which are two totally different ways to describe the dynamic aspect of a system.

3.2. Real-time Events and Cycle-based Events

UML-RT offers state machines inspired from Harel's statecharts [8] to model the behavior of a capsule. These state machines have the following characteristics:

- Concurrent: state machines in different capsules can be executed concurrently.
- Reactive: transitions occur when an event is raised on the capsule's communication ports.
- Hierarchical: state machines can be nested to simplify behavior modeling

Through code generation, these state machines become executable software, where events are operation calls processed in real-time. Although a UML-RT state machine has many advantages to model a system's dynamic aspects, some caveats are observed when interacting with components described in a cycle-based environment. For instance, some cycle-based components rely on registered inputs that are sampled on clock ticks (typical to the discrete-time and discrete-events models of computation). In UML-RT state machines, information is transferred during punctual operation calls and therefore the communication link retains no information. On the other hand, toggling a signal in a cycle-based system does not generate an event by default: an observer must listen passively on the line to capture the state of the signal and react upon a change (defined as the sensitivity of a process). In UML, state machine transitions are actively triggered by a calling system in a one-shot manner.

The test infrastructure will manifest itself to the executable model under a cycle-based form. Since the primary role of the test infrastructure is to represent the world outside, it will include the role of the analog-to-digital and digital-to-analog processing happening on the outskirts of the SoC. On a smaller-scale, the test infrastructure can also act as some clocked co-processor available in an IP library and used during system refinement. In either case, it is safe to assume that the real world will communicate with the executable model via clocked samples.

To enable both worlds to communicate efficiently, cycle-based data must translate to active events interpreted by the model's statecharts. We will present a communication mechanism based on a simple client-server model that solves this problem.

4. Role of Transactors in Prototyping

4.1. Model-Level Transactors vs. Transaction-Level Modeling

It is first important to define what a transactor is in the context of our work, and how it relates to other usage of the concept. A transactor enables communication between *simulated-time* and *real-time* components. Model-level transactors are used in the UML-RT model as capsules that communicate with clocked or cycle-based models running in simulators. Figure 2 depicts the location of transactors in the functional prototype.

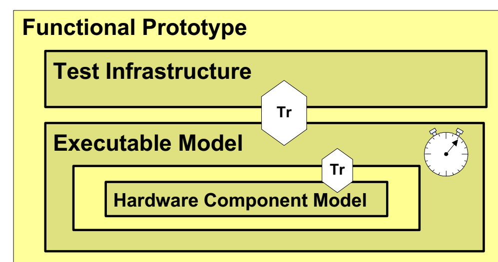

Figure 2. Location of transactors (Tr)

IEEE COMPUTER SOCIETY

In SystemC, transactors are used for transaction-level modeling and verification. They enable transactions from one level of abstraction to the other, inside the same simulated-time environment. A SystemC transactor is a hardware device that, on the software side, communicates at the transactional level, and, on the hardware side, at the signal level [9].

This usage of transactors is very hardware-biased, as are models issued from SystemC. Moreover, these transactors are dependent on the SystemC scheduler that controls the timing and order of process execution [10]. By contrast, model-level transactors are high abstraction entities with built-in synchronization standing between real-time and simulated time domains. Their implementation handles transactions between a working prototype, running in real-time without the support of a simulator, scheduler or ISS, and another component running under one of these constraints.

As a side note, transactors do not exist in the final implementation of the system. Iterative refinement will transform them into one of the following forms that are relevant in systems-on-chip:

- drivers or interrupt service routines for software-hardware interaction;
- bus slaves and masters or custom signals for hardware-hardware interaction;
- Application Programming Interface (APIs) for software-software interaction.

4.2. Client-Server Model

A server is an entity that responds to a client's requests. Likewise, an embedded application is a processing entity offering a service to clients. If the processing entity is disabled, client requests are denied. If client requests stop, the server will go idle or do some internal business. In this paper, the server is the executable model and the client is the cycle-based test environment. From a hardware refinement perspective, the executable model becomes the client of multiple hardware accelerators seen as servers. This case is a conjugate of the first one and will not be covered here.

The transactor that will enable the communication between these two entities is divided into a server half and a client half that live in the executable model and the cycle-based model respectively. Of course, each half is itself divided in a front-end that is compatible with the model and a back-end that implements the client-server signaling. From this standpoint, different communication scenarios are described in Table 1.

Dataflow synchronization between cycle-based and event-based domains relies on the identification of atomic processes in the capsule's state machines. An atomic process is a chain of events occurring between two interactions of the executable model with the clocked model. These interactions may occur every cycle (as assumed in this paper) or span across n clock periods. Determining the value of n can be used as a partitioning metric, which is left as a topic of future research.

4.3. Implementing Transactors with Sockets

The Berkeley sockets are an efficient communication mechanism that can be used to implement the back-end of a model-level transactor. On the UML side, the transactor is a capsule connected to the ports of the design, hiding the socket implementation from the model. On the client side, the same socket mechanism is implemented using a client compatible front-end. In Simulink, for example, the client transactor is implemented in a user-defined block called a C++ S-function. This is a special block that can be inserted in a Simulink model to execute custom C++ code during a simulation. When the Simulink scheduler executes the S-Function block, the custom code is executed and all the transactions with the UML-RT model relevant to this cycle are performed. The characteristics of this socket mechanism are:

- Communication between the executable model and the clocked component is synchronous and interactive. A socket can act as a signal observer similar to a latch, enabling fine resolution feedback and realistic execution scenarios.
- The communication overhead is still significant from a real-time perspective, and therefore the capsule transactor must be easy to remove for proper model refinement.

Table 1. Communication Scenarios between Event-Based (Server) and Cycle-Based (Client) Behavior

Flow of communication	Initiator	Typical Situation	Example
1. Client → Server	1.1 Server (with handshaking)	When a server requires information from a client, it sends a message on its transactor. A client transactor is observing this trigger on the cycle-based side. The information is sent back to the server	UML capsule ready to receive data from Simulink environment
	1.2 Client (no handshaking)	Upon a cycle tick, the client transactor checks for a change in a given signal state. If a change is detected, a message is sent to the server through the transactor. The transaction requires polling or interruption on the server side.	Reset signal sent from Simulink to UML
2. Server → Client	2.1 Server (no handshaking)	Data is sent to the client transactor that monitors the line. The cycle at which data is expected must be known precisely, or an observer must be implemented on the client side.	Processed data returned back to the test environment after n client cycles
	2.2 Client (with handshaking)	When the conditions apply, the clocked module that needs data activates a request line on the transactor. A request event is sent to the server and data is returned at a known time.	Simulink retrieves the value of a state register in the executable model

IEEE COMPUTER SOCIETY

Sporadic signals such as a reset signal expose a limit of a UML-RT model. The delay produced by the translation of the toggle on the line into an event for the capsule state machine is less deterministic than in hardware (c.f. scenario 1.2 in Table I). The granularity of events in the capsule world is a lot coarser than in the cycle-based world and it is the role of the transactor to act as a buffering and recovery mechanism. This leads to an important observation in the usage of UML-RT for integrated systems: a statechart is a high-level construct that cannot profitably imitate the timing granularity of clocked hardware components. Knowing this limit, unnecessary modeling efforts can be avoided.

5. Case Study

5.1. Prototyping of a Software Radio Component

Model-level transactors have been used in the functional prototyping of an adaptive equalizer for a software radio. The prototype will be presented here as a case study. The tool used to build the executable model is Rose-RT from IBM-Rational, which offers efficient model-to-code translation. The UML-RT state machines are detailed using C++ code, which is essential to obtain a complete behavioral model. The generated C++ code is compiled into an executable file and runs on a host station.

The test infrastructure is developed in parallel with Simulink to produce test stimuli and analyze test results. Simulink has numerous pre-built components to generate and analyze radio signals, like QPSK encoders, channel models and constellation scope. The test infrastructure and the executable model of the equalizer, coupled with the appropriate transactor, constitute the first version of a functional prototype, which was up and running in a matter of days.

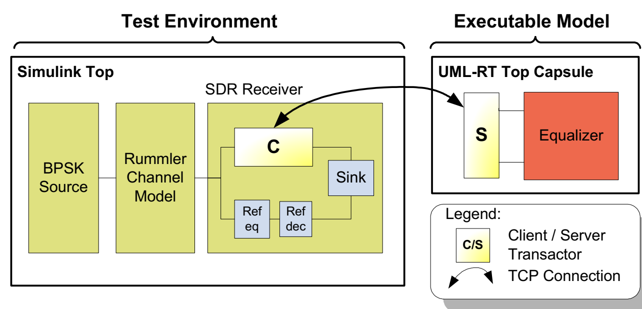

Figure 3. Functional prototype built with Simulink, UML-RT and transactors

5.2. Experimental Results

The adaptive equalizer has been completely specified using UML-RT capsules, state machines and C++ code realizing the dataflow. Its development is influenced by the fact that the equalizer will eventually be part of a larger software-defined-radio system. A complete test environment has been set up in Simulink by a different team to produce realistic data and to analyze the processing results. Figure 3 presents the block diagram of the complete functional prototype.

The execution of this functional prototype is a two-step process. The UML-RT model is first launched to act as the server. The transactor capsules are created and the server completes its initialization procedure. Then the Simulink test environment is launched and its client transactor issues a connection request to the server transactor. Once the connection is established, the Simulink fixed-step solver starts the cyclic execution of the test environment blocks, producing the test data we need. The diagrams of Figure 4 illustrate the structure of the UML-RT half of the transactor and its connection to the rest of the model.

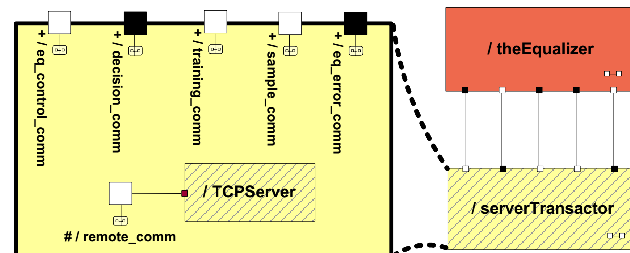

Figure 4. Key UML structure diagrams

On the Simulink side, the client transactor is implemented in a user-defined S-function block, as shown by the model snapshot of Figure. 5.

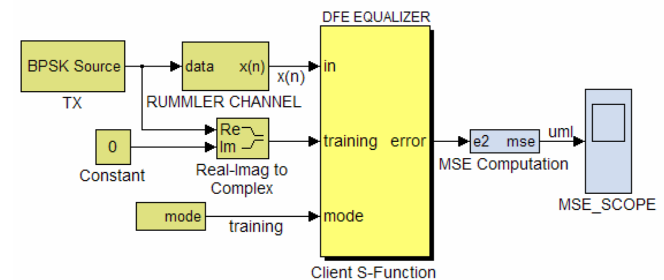

Figure 5. Test environment in Simulink

A reliable solution to interconnect the client and server halves of the transactor is to implement basic handshaking between the clock-based and event-based domains (c.f. scenario 1.1 in table I). To this end, a *ready* signal is sent by the executable model when it is ready to receive new data. Blocking TCP sockets were needed to ensure proper synchronization, because of the sensitivity of the prototype to variable speed of execution on different hosts. Indeed, on one type of computer, Simulink waited at each cycle for the executable model to send the *ready* signal as illustrated in scenario 1 of Figure 6. On a faster computer though, the model ran faster than a Simulink cycle, producing a “ready” signal before the completion of the Simulink cycle, as illustrated in scenario 2. In the first scenario, the clocked transactor socket blocks the execution of Simulink until the “ready” signal is received. In the second scenario, the same socket keeps the “ready”

IEEE COMPUTER SOCIETY

signal in a queue until it is sampled at the beginning of a cycle. It is interesting to note that only the first scenario will provide continuous real-time bit rates, since the executable model runs at its maximum speed. In the second scenario, the bit rate is artificially slowed down by the unrealistic execution time of Simulink, which prevents measurement of a continuous real-time bit rate.

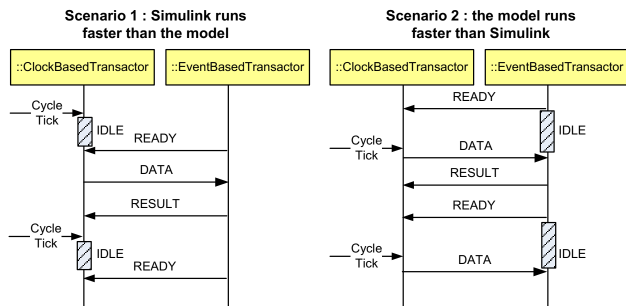

Figure 6. Execution scenarios on two different computers

The execution of this functional prototype on a Pentium 4 CPU running at 1.7 GHz on a non real-time OS has allowed us to reach a sample period of 32 µs for basic BPSK modulation, or 62.5 kbps. Although the code could be optimized to reach a higher rate or the modulation changed to higher density, this rate would be enough to handle digital voice communication over a radio channel.

Figure 7 was traced dynamically by Simulink with data gathered on the fly from the executable model. It represents the evolution of mean square errors (MSE) vs. time in an adaptive equalizer. This kind of analysis was extremely useful in the validation and refinement of the specification of the system.

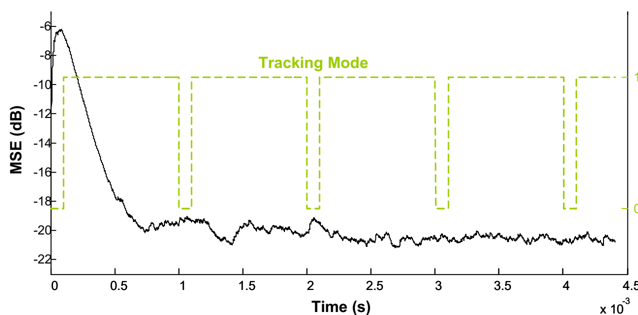

Figure 7. Mean-square error vs. time

6. Observations and Conclusion

Transactors allow breaking tool barriers and implementing functional prototypes that can be tested in a virtual simulated environment, and then run under realistic conditions. To achieve an accurate stimulation of the system, a Simulink model was used as a test environment to produce realistic radio channel data. The coupling of a UML-UML model to the Simulink tool with model level transactors allowed a functional and inter-active prototype of a system to be built.

The hardware implementation components that can be introduced in a UML-RT model will eventually use different communication means such as an AMBA-AHB bus, interrupt pins or custom protocols. Somehow, the transactor should be transformed into one of these hardware implementations, like a bus protocol. Modeling such communication infrastructure is left for future investigation. Another interesting issue is the exchange of fixed-point data with clocked components. Fixed-point data format are used in hardware systems where data lines have finite widths. The translation from fixed-point to floating point is another functionality which could be implemented in the model-level transactors.

This work opens a variety of opportunities to bring high-level modeling and functional prototypes closer to hardware implementations.

7. Acknowledgements

Academic licenses of tools provided by IBM and Mathworks enabled this research. Philippe Dumais, Stéphane Cormier, Olivier Munger, members of the MAME team at École de Technologie Supérieure in Montréal, developed the Simulink test environment under the guidance of Professor François Gagnon, team leader of the Prompt project that supports this research.

8. References

[1] E. Kemp, D. Pacitto, E. Todd, D. Gray, *The role of functional prototyping in model validation*, in Proc. of the 1996 Information Systems Conference of New Zealand, IEEE Computer Society Press, Los Almos, California, 1996.

[2] N. Habra, *A transformational method for functional prototyping*, IEE Colloquium on Automating Formal Methods for Computer Assisted Prototyping, London, U.K., 1992.

[3] L. Lev, R. Razdan, C. Tice, *It's about Time: Requirements for the Functional Verification of Nanometer-Scale ICs* [online], Cadence White Paper, 2003, available from <http://www.cadence.com/whitepapers/ 4451_IncisiveWP_FNL.pdf>

[4] OMG, *MDA Guide Version 1.0.1,* ed. J. Miller and J. Mukerji, document number omg/2003-06-01, 2003.

[5] K. McGroddy et al., *SOC – The IBM Microelectronics Approach*, Chapter 6, Winning the SoC Revolution, Kluwer Academic Publishers, pp. 119-140, 2003.

[6] OMG, *UML Profile for Schedulability, Performance, and Time Specification*, Version 1.0, September 2003.

[7] Rational Rose RealTime, ver. 2003.06.00.436.000, Rational Software Corporation, Cupertino, Calif.

[8] D. Harel. *Statecharts: A visual formulation for complex systems.* Science of Computer Programming, June 1987.

[9] Cedric Alquier Stephane Guerinneau Lauro Rizzatti Luc Burgun, *Co-Simulation Between SystemC and New Generation Emulator*, DesignCon 2003.

[10] OSCI, *SystemC 2.0.1 Language Reference Manual*, Revision 1.0, 2003.

[11] Presentations of the *UML for SoC Design Workshop* at DAC'04, available from <http://www.c-lab.de/uml-soc/>

[12] Overview of the Ptolemy Project, Technical Memorandum UCB/ERL M01/11.

[13] P. Bjuréus, A. Jantsch, *MASCOT: A Specification and Cosimulation Method Integrating Data and Control Flow*, DATE 2000.

[14] Seamless, Mentor Graphics Corporation, Wilsonville OR.