Executable system-level specification models containing UML-based behavioral patterns

Leandro Soares Indrusiak, Andreas Thuy, Manfred Glesner
Institute of Microelectronic Systems - Technische Universität Darmstadt
E-mail: <indrusiak, glesner>@mes.tu-darmstadt.de, a.thuy@gmx.net

Abstract

Behavioral patterns are useful abstractions to simplify the design of the communication-centric systems. Such patterns are traditionally described using UML diagrams, but the lack of execution semantics in UML prevents the co-validation of the patterns together with simulation models and executable specifications which are the mainstream in today's system level design flows. This paper proposes a method to validate UML-based behavioral patterns within executable system models. The method is based on actor orientation and was implemented as an extension of the Ptolemy II framework. A case study is presented and potential applications and extensions of the proposed method are discussed.

1 Introduction

Much of our capacity to design and optimize systems depends on our ability to recognize patterns within design problems and compare them with other problems we already know how to solve. Based on such premise, the software engineering community started in the late 90's to identify and catalogue well known solutions to recurrent problems in object-oriented software development, calling them design patterns. Flagships of such initiative include the seminal book by Gamma et al [1] and the Portland Pattern Repository [2]. While not particularly original, the idea of documenting patterns and facilitating their reusability was well accepted and surpassed the limits of object-oriented software engineering, as evidenced by applications in SoC design [3], real time systems [4], reconfigurable computing [5] and wireless sensor networks [6].

According to Gamma [1], patterns can be classified regarding their purpose: creational, structural or behavioral patterns. Creational patterns address the process of object creation, while structural patterns deal with the composition of classes or objects and behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility. Traditionally, such patterns have been described using UML diagrams. In [1], all patterns are illustrated by UML class diagrams and many of the descriptions of behavioral patterns also include sequence diagrams covering the dynamics of the message exchange among the participants of the pattern.

In this paper, we advocate for the usage of behavioral design patterns (referred throughout the text as behavioral patterns) as a means to foster reusability of communication schemes within system-level specification models. In order to keep with the *de facto* standard for behavioral pattern representation and also to benefit from legacy patterns, we will use standard UML diagrams in our approach. However, in order to maximize the potential usage of such patterns, we propose to integrate them within executable and simulatable specification models (such as Matlab/Simulink, SystemC, VHDL and Verilog), which are mainstream in today's system level design flows. By advocating for such integration, our approach differentiates itself from the related work in UML-based SoC specification, covered in Section 2 of this paper. The proposed integration is based on the actor-oriented paradigm [7], which is detailed in Section 3. The proposed integration strategy is covered by Section 4, and a case study implemented as an extension to the Ptolemy II actor-oriented framework is reported in Section 5. The paper is closed with a discussion of the achieved results and potential extensions in Section 6.

2 Related Work

Most of the current approaches using UML as a hardware/software system-level specification language address either static analysis or code generation methods. Oliveira et al [8] use static analysis to evaluate performance, memory footprint and power consumption of alternative embedded software behavioral patterns modeled using UML sequence diagrams. In [9], UML class diagrams are used as templates for design space delimitation and exploration.

Different alternatives for code generation techniques based on UML diagrams can be found in a number of research initiatives (mainly driven by industry). Many of

978-3-9810801-2-4/DATE07 © 2007 EDAA

them advocate for the joint usage of UML and SystemC. In [10], a team of researchers from University of Catania and ST Microelectronics stated that UML should be seen as a high level modeling language and SystemC as a low level system language. In order to allow the interoperability of both languages, they proposed a set of stereotypes that allow the modeling of SystemC concepts using UML diagrams. The stereotypes were grouped together in the so-called UML 2.0 profile for SystemC. Similar approaches – with different strategies on the definition of stereotypes – were presented by researchers from Politecnico de Milano and Siemens ICM [11] and Fujitsu [12]. In all three cases, the stereotypes were created to support code generators, which should be able to generate SystemC code out of UML models. Code generation is also explored in [3] and [13]. The former targets the automatic generation of component wrappers according to UML models, while the later addresses the modeling and code generation targeting a FPGA-based execution platform.

The proposed method differs from all the forementioned approaches because it allows the validation of UML diagrams together with additional system/subsystem models (legacy models, hardware-in-the-loop, etc.). Static analysis of UML models can only provide information that may aid the designer on constructing the actual design model. Code generation approaches require radical model transformation in order to allow the validation of the system specification using UML. The proposed approach, however, supports the direct validation of UML models by including them within executable and simulatable system models. This allows for a more realistic design flow, because it is very unreasonable to expect that a system could be completely modeled in UML alone, and its implementation will be completely generated *a posteriori* in a pure model-driven fashion. The most likely scenario, which is the target of the approach presented here, allows for heterogeneous specifications in different languages and different levels of abstractions. In such scenario, UML can be used only when it is the best suitable method (for instance, to model behavior patterns).

3 Actor-oriented System Specification

The inclusion of UML diagrams within executable and simulatable models is not straightforward. Firstly, because different types of executable and simulatable models can present significant differences in syntax and semantics. Secondly, because UML comprehends a number of different diagrams focusing on different views of a system, and each of them will interact with an executable specification in a different way. The approach presented in this paper tackles the two difficulties by restricting (a) the type of executable models it supports and (b) the types of UML diagrams it uses. The first restriction, which limits this approach to actor-oriented models, will be justified as follows, while the restriction on the types of UML diagrams will be covered in Section 4.

Actor orientation was initially proposed as a model for concurrent computation by Hewitt and further developed by Agha [14]. It is based on actors, which are concurrent elements that communicate through asynchronous message passing. More recently, the model was revisited by Lee within his work on the Ptolemy II framework [7]. Lee's approach supported the implementation of heterogeneous actor-oriented models, allowing experimentation with different models of concurrency and communication in a single system model through hierarchical composition.

To support such heterogeneity, Lee separated completely the syntax and the semantics of the models. The model structural features are represented by the so-called abstract syntax, which comprehends the actor placeholders, their input and output *ports* and the *relations* that establish connections between ports (Figure 1). The actor placeholders are generic enough to encapsulate actors specified in different languages, which actually provide the particular semantics of each actor. In the Ptolemy II framework the actors can be specified using Java, C/C++, Cal, Matlab, Python, among others. Available extensions to that framework are known to support also VHDL, Verilog, SystemC [15] and even FPGA-based hardware-in-the-loop modules [16].

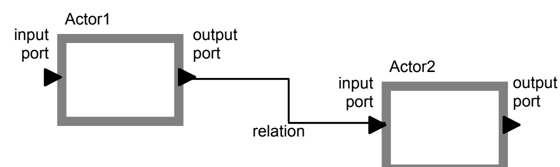

Figure 1. Actors, ports and relations in Ptolemy II

For the purposes of this paper, however, the major feature of Lee's view on actor orientation is the possibility to compose heterogeneous models hierarchically. Such approach allows different execution semantics to be associated to each hierarchical level of the model, in opposition of the commonly used global scheduler. This means that actors can be contained inside of other actors (so-called *composite actors*), and the inner actors are allowed to inherit the execution semantics from its container or to be assigned their own. The execution semantics, which defines how and when each actor can communicate, compute and update its internal state, is defined by a *director*. Many directors were implemented within Ptolemy II, representing well known concurrent

models of computation (MoCs) such as discrete events, synchronous dataflow and Kahn process networks.

In summary, actor orientation within Ptolemy II allows for experimentation on concurrent execution of actors, which can be specified in a variety of languages or as a composition of other actors, and in each hierarchical level of composition a new definition of execution semantics can be considered. In the following section, such features are used to allow the inclusion of actor-encapsulated UML diagrams representing behavioral patterns within Ptolemy II models.

4 Integrating Behavioral Patterns to Actor-oriented Models

Behavioral patterns are traditionally described using UML class and sequence diagrams. Class diagrams define the types of participants of the patterns and which messages each of them can receive, while sequence diagrams detail the precedence and concurrence relationships among the exchanged messages. A simple example is shown in Figure 2, representing a distributed consensus pattern according to the Chandra-Toueg algorithm [17], which is widely used in fault-tolerant distributed systems. The class diagram in Figure 2a shows the two participating classes - Coordinator and Process – as well as their interrelation (generalization) and the message signatures for each of them. The dynamics of the pattern is shown on Figure 2b, where the vertical lines represent the lifelines of three instances of the classes shown in the corresponding class diagram, and the arrows represent asynchronous message calls between them.

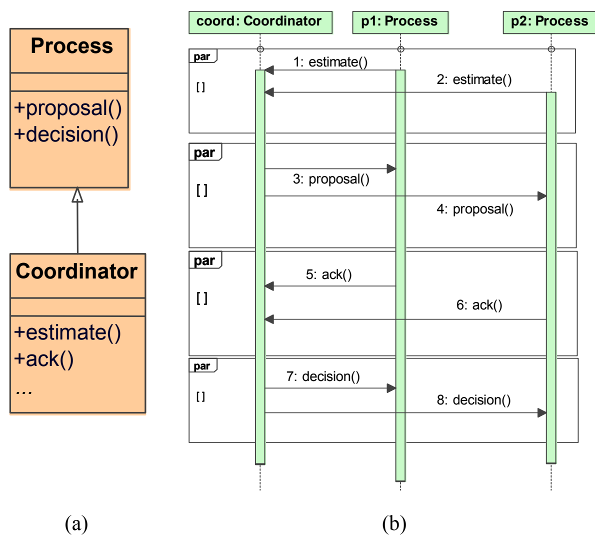

Figure 2. Behavioral pattern in UML

The correspondence of the modeling constructs from UML to those in actor-oriented models is somehow straightforward, but can give margin to distinct interpretations. In this paper, we follow an initial approach from [18] and start making the following assumptions for the integration of sequence diagrams into actor-oriented models:

(1) all sequence diagrams are contained by a composite actor;

(2) each instance of a class represented as a lifeline in a sequence diagram is statically and bijectively mapped to an actor contained by the composite actor mentioned in assumption 1;

(3) each message between instances in a sequence diagram is statically mapped as ports and relations into the corresponding actor, as follows:

> (3.1) asynchronous messages are mapped to an output port in the actor associated to the instance on the sending side of the message, an input port in the actor associated to the instance on the receiving side of the message, and a relation between them;
>
> (3.2) synchronous messages are mapped the same way as defined for asynchronous communications, additionally to an input port in the actor associated to the instance on the sending side of the message, an output port in the actor associated to the instance on the receiving side of the message, and a relation between them, in order to allow the return of the control flow to the sending actor which must be suspended upon the synchronous call.

Figure 3 illustrates the application of the basic set of assumptions. Figure 3.a shows a simple sequence diagram with one synchronous and one asynchronous message, which are encapsulated within the composite actor shown in Figure 3.b.

Some of these assumptions were already taken into account on two different approaches for the integration of sequence diagrams and actor-oriented models reported in [18] and [19]. In [18], the sequence diagram was integrated as a composite actor that uniquely schedules the message calls to its ports according to the sequence diagram it encapsulates. The mapping between lifelines and actors is done by linking the ports of the actors with those ports associated to the message calls (Figure 3.c). Differently, in [19] the sequence diagram is implemented as an attribute to the director associated to a given composite actor. It means that the sequence diagram lifelines are created in accordance to the actors contained within the composite actor, and the sequence of messages can be either enforced or verified by the director.

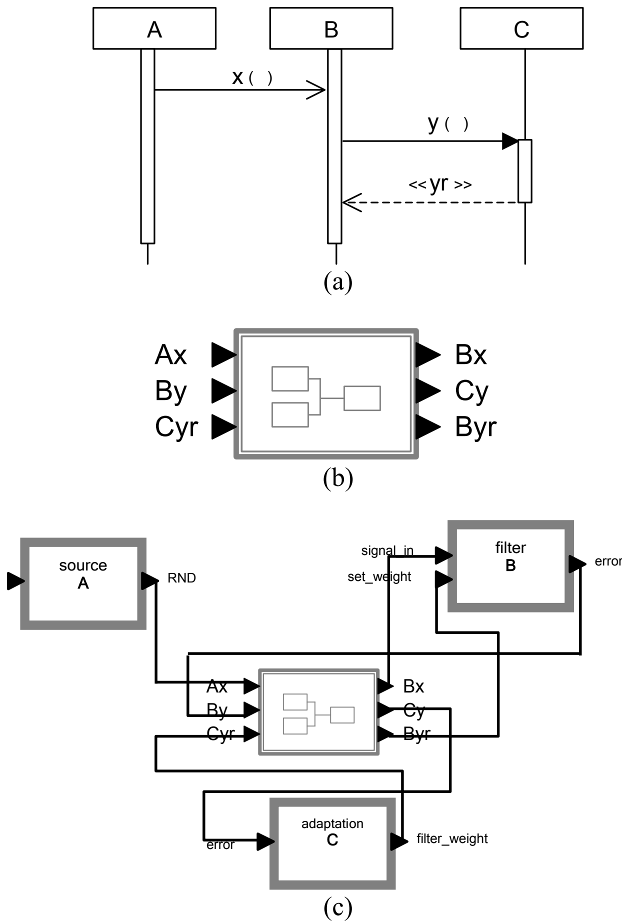

Figure 3. Integration of sequence diagrams and actor-oriented models

While such assumptions are sufficient for allowing the inclusion of simple behavioral patterns within actor-oriented models, there are still some limitations:

- the structure of the pattern can't be completely modeled, as only the signatures of (part of) the messages can be derived from the sequence diagram. No definition of types are explicitly modeled, so nominal subtyping can't be explored and only a limited form of structural subtyping is available;

- the mapping between instances and actors in assumption 2 prevents dynamic polymorphism, which may be required in a number of behavioral patterns. For instance, in the example of Figure 2, Coordinator inherits from Process (denoted by the generalization relationship), which means that every instance of Coordinator is also an instance of Process. Such relationship is actually needed in the actual execution of the Chandra-Toueg algorithm, because a different process takes up the role of coordinator on every consensus round. Following the basic set of assumptions, the complete implementation of the algorithm using the pattern shown in Figure 2 would not be possible, because a single actor would be permanently mapped to the coordinator role.

In order to overcome the limitations, we update the set of assumptions as follows:

(2) each instance of a class represented as a lifeline in a sequence diagram is statically or dynamically mapped to an actor contained by the composite actor mentioned in assumption 1. The dynamic mapping function must observe assumption 5;

(4) class diagrams can be included as attributes to composite actors; lifelines in a sequence diagram can be assigned a class or interface from those available as attributes on its container, or recursively up to its top-level container; ports in an actor can be assigned a class or interface from those available as attributes on its container, or recursively up to its top-level container. For input ports, the assignment denotes the required interfaces of that port, while for output it denotes the provided interface, in a similar fashion as in a UML 2.0 composite entity diagram;

(5) dynamic mapping of lifelines and actors is conditional to type checking, which means that a lifeline will only be associated to an actor that was assigned a type (class, interface) which is the same, or a subtype, of its own.

By relying on the extended set of assumptions, system specification can be done using three types of UML diagrams – class, sequence and composite structure – which are fully integrated to an actor-oriented model.

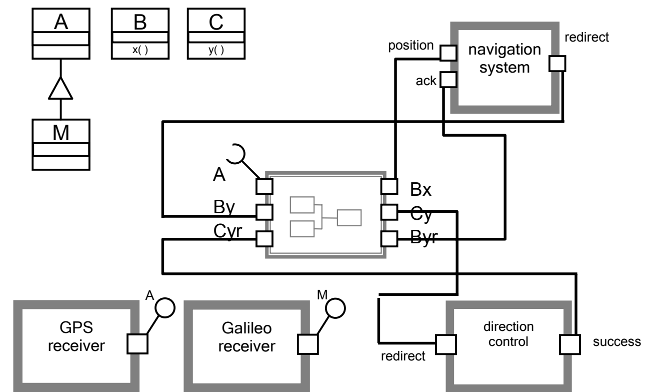

Figure 4. Actor-oriented model including encapsulated sequence diagram and explicit interface definitions as attributes

Figure 4 depicts a new usage of the sequence diagram from Figure 3.a, but this time exploring the extended set of assumptions. Notice the class diagram appearing as an attribute of the model in the upper left side, so that the type and subtype definitions are done explicitly and

nominally. The model itself, which is still an actor-oriented model, now follows the visual syntax of a composite structure diagram, with decorators for provided and required interfaces on the ports. The presented example also includes static relations between actors and lifelines through port connections as in the example of Figure 3, but in this case both “GPS Receiver” and “Galileo Receiver” actors can be dynamically associated to the lifeline A embedded within the composite actor, because both satisfy the interface requirement of the corresponding port (“GPS Receiver” provides interface A, while “Galileo Receiver” provides interface M, which is a subtype of A).

5 Case study

The validation of the proposed assumptions described in the previous section was partially achieved with the implementation of an extension to the Ptolemy II framework. Additional constructs were added to the Ptolemy data model and corresponding user interface components were added to Vergil (Ptolemy II GUI) in order to allow the entry of sequence diagrams by the user and the correct execution of the modeled communication behavior. Extensions to the Ptolemy II director library were also needed, in order to support the sequential dependency constraints imposed by the sequence diagrams.

The extended framework was used to model the behavioral pattern from Figure 2. It was encapsulated within a composite actor according to the approach described in the previous section. To verify its functionality within an application scenario, we modeled a testbench using Ptolemy II actors representing a distributed system composed of three sensor nodes observing a particular feature (for instance, the amount of infrared radiation in a given location or the license plate of a car in a highway). The three nodes use the distributed consensus pattern to check if their readings are consistent (in normal operation, all three readings must be the same). In the modeled testbench, each node is associated to a process, one of them being the coordinator. The sensors send data to the processes periodically, and the processes use the behavioral pattern to obtain consensus regarding the read value. The implemented actor-oriented model is shown in Figure 5, where the larger actor on the center-right side encapsulates the UML sequence diagram.

The execution of the model showed the correct functionality of the sequence diagram, which enforced the proper exchange of messages between the coordinator and the two other processes, and the consensus was correctly achieved. In order to validate the modeled system under more realistic circumstances, we included two types of faults: delays of the sensor readings and faulty readings. The first fault represents the non-deterministic time intervals which can arise, for instance, due to image processing or data compression functionality at the sensor side, while the second models a transient fault that affected the sensed value itself.

The delay was modeled as a uniformly distributed variable, while the sensor reading error was modeled as a poisson process characterized by a mean time between faults (MTBF). Table 1 shows the obtained simulation results for the number of failures to achieve consensus in a batch of 1000 readings per sensor (one reading per sensor per time unit) under different reading delays and different MTBFs. Each result was obtained by averaging the number of failures obtained in 10 simulation runs.

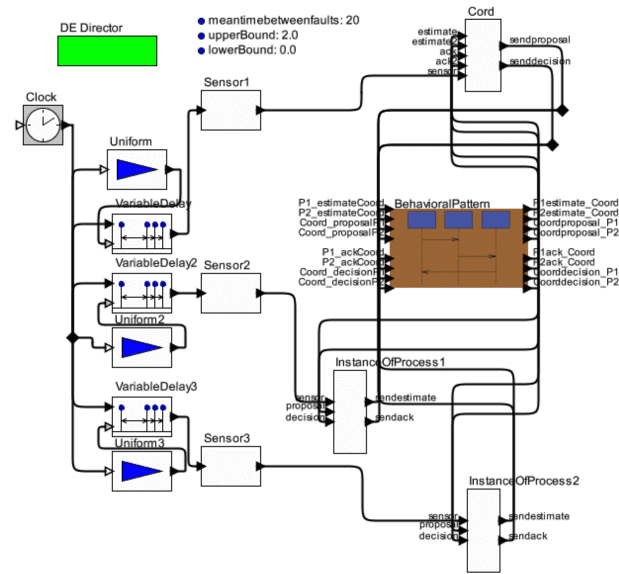

Figure 5. Case study in Ptolemy II

delay	MTBF				
	20	30	40	50	60
0-0.2	6.4	2.5	1.1	0.0	0.0
0-2.0	6.0	2.7	0.3	0.0	0.0

Table 1. Failures to achieve consensus among sensor readings

It is possible to conclude that the system is reliable when the mean time between faulty sensor readings is larger than 50 time units, as the consensus pattern worked 100% of the times in such scenario, even under presence of faults in one of the sensors (the algorithm is designed to tolerate $f < n/2$ faults). It is also important to notice that the behavioral pattern ensured the correct communication behavior even under delays of the same order of magnitude as the sensor throughput.

6 Conclusions and future work

This paper presented an approach to model and execute UML-based behavioral patterns together with actor-oriented system specifications. The approach differs from related work by relying on the joint execution of UML-based patterns and actor models. Such integrated approach has the potential to increase the acceptance of UML within system-level designers by allowing them to explore only the features of UML which can bring immediate benefit to the design methodology they are already familiar. Instead of learning a new (and relatively complex) language from scratch, they can learn some of the features of UML and understand it from the point of view of a classic block-based modeling and execution framework.

In order to bring this approach into practice, a number of assumptions were made to link basic UML concepts to actor-orientation constructs, and such assumptions were used as guidelines to extend a well-known framework – Ptolemy II – so that designers can graphically enter executable system specification models containing UML-based behavioral patterns. A case study was implemented on top of this framework, modeling a distributed consensus pattern applied to a distributed sensor system. Simulation results were obtained on the robustness of the system under two types of faults: sensor reading delays and faulty readings.

The presented extension to Ptolemy II is still work in progress, as it doesn't implement completely the extended set of assumptions presented in Section 4. For instance, the possibility of dynamic mapping between actors and lifelines in a sequence diagram was not yet fully explored. Such possibility, to be addressed in future work, is of particular interest for the modeling of emerging behavioral patterns in ad-hoc wireless networks (patterns which are formed dynamically when nodes of particular types are in each other's neighborhood) or in multi-processor systems containing reconfigurable processors (processors can reconfigure their datapath dynamically in order to implement a given type which is needed to establish a desired pattern). Furthermore, the presented techniques would also benefit from a formal approach to the mapping of UML and actor orientation concepts at the metamodel level.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley, 1995.

[2] H. G. Cunningham et al., “Portland Pattern Repository”, January 2006, http://c2.com/ppr .

[3] R. Damaševičius, G. Majauskas, and V. Štuikys, “Application of design patterns for hardware design,” in Proc. 40th Design Automation Conf., 2003, pp. 48-53.

[4] B. P. Douglass. *Real-Time Design Patterns: Robust Scalable Arch. for Real-Time Systems*. Addison Wesley, 2003.

[5] André De Hon et al, “Design Patterns for Reconfigurable Computing”, in Proc. IEEE Symp. Field-Programmable Custom Computing Machines, 2004, pp. 13-23.

[6] D. Gay, P. Lewis, and D. Culler, “Software design patterns for TinyOS,” in Proc. ACM SIGPLAN/SIGBED Conf. on languages, compilers, and tools for embedded systems (LCTES), 2005, pp. 40-49.

[7] E. A . Lee, S. Neuendorffer, and M. J. Wirthlin, “Actor-Oriented Design of Embedded Hardware and Software Systems,” *Journal of Circuits, Systems, and Computers*, vol. 12, n. 3, pp. 231-260, 2003.

[8] M.F. S. Oliveira, L. Brisolara, F.R. Wagner, L. Carro. “Embedded SW Design Exploration Using UML-based Estimation Tools,” presented at DAC Workshop on UML for SoC Design (UML-SOC), Anaheim, USA, 2005.

[9] L. S. Indrusiak, M. Glesner, M. E. Kreutz, A. A. Susin, and R. A. L. Reis, “UML-Driven Design Space Delimitation and Exploration: A Case Study on Networks-on-Chip,” in Proc. IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 2004, pp. 5-12.

[10] E. Riccobene, P. Scandurra, A. Rosti, S. Bocchio. “A SoC Design Methodology Involving a UML 2.0 Profile for SystemC,” in Proc. of IEEE/ACM Design Automation and Test in Europe, 2005, pp. 704-709.

[11] F. Bruschi, D. Sciuto. “A SystemC based design flow starting from UML models,” presented at European SystemC Users Group Meeting (ESCUG), Lago Maggiore, Italy, 2002.

[12] Q. Zhu, R. Oishi, T. Hasegawa, T. Nakata. “System on Chip Validation using UML and CWL,” in Proc. of IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis, 2004, pp. 92 – 97.

[13] T. Schattkowsky, W. Mueller, and A. Rettberg, “A Generic Model Execution Platform for the Design of Hardware and Software” in *UML for SOC Design*, G. Martin and W. Müller, Eds. Dordrecht: Springer, 2005, pp. 63-88.

[14] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Cambridge: MIT Press, 1986.

[15] Mirabilis Design Inc. „VisualSim“, July 2006, http://www.mirabilisdesign.com/ .

[16] D. F. Jimenez-Oróstegui, L. S. Indrusiak, and M. Glesner, "Proxy-Based Integration of Reconfigurable Hardware Within Simulation Environments". In Proc. IEEE Int. Conf. Microelectronic Systems Education, 2005, pp. 59 – 60.

[17] T. D. Chandra, S. Toueg, „Unreliable Failure Detectors for Reliable Distributed Systems,“ Journal of the ACM , 43(2), pp. 225-267.

[18] L. S. Indrusiak, A. Thuy, and M. Glesner, „On the Integration of UML Sequence Diagrams and Actor-Oriented Simulation Models,” presented at DAC Workshop on UML for SoC Design (UML-SOC), Anaheim, USA, 2005.

[19] A. Thuy, L. S. Indrusiak, and M. Glesner, „Applying Communication Patterns to Actor-Oriented Models with UML Sequence Diagrams“, in Proc. Forum on Spec. and Design Languages, 2006.