Towards Design Method Based on Formalisms of Petri Nets, DEVS, and UML

Radek Kočí and Vladimír Janoušek

Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
{*koci,janousek*}*@fit.vutbr.cz*

***Abstract*—Software system development uses specific development techniques and processes to reach desired goals, whereas different kinds of systems usually need to use different approaches. Obviously, there are used different techniques, tools, and formalisms in each development process and the designed models should be automatically or manually transformed to the next development step. The paper is aimed at such development processes, which work with formalisms allowing to design architecture and functionality, analysis of design, testing and system run with no need to change this formalism. Nevertheless, there can be useful to combine more different formalisms and model languages because of developers are used to use these formalisms or there are already created models using these formalisms. The paper deals with UML, Petri Nets, and DEVS application in the systems design and sketches a method how to use the formalisms for modeling a system architecture and its behavior. Its combination decreases a number of transformations of models and makes the architectural description well-arranged.**

***Keywords*-Simulation-Based Design, Object-Oriented Petri Nets, DEVS, UML.**

I. INTRODUCTION

The key activities in the system development are specification, testing, validation, and analysis (e.g., of performance, throughput, etc.). Most of the methodologies use models for system specification, i.e., for defining the structure and behavior of developed system. There are different kinds of models, from models of low-level formal basis to pure formal models. Each kind has its advantages and disadvantages. The most popular modeling language in software engineering is UML [1]. It serves as a standard for analytics, designers and programmers. But, own phraseology of UML does not have enough power allowing to realize some fundamental relationships and, in particular, rules, that are branch of every modeled system. For example, how can we define a condition that at least one item has to be at a stack when the operation *pop* is called? The pure UML language does not offer suitable tools. Although the UML language can be completed by OCL (Object Constraint Language), stereotypes, etc., which makes the system description more precise, it makes the checking of system correctness or validity by means of testing or formal methods very complicated.

Therefore, the new methodologies and approaches are investigated and developed for many years. They are commonly known as Model-Driven Software Development or Model-Based Design (MBD) [2], [3], [4]. An important feature of these methods is the fact that they use executable models, e.g., Model Driven Architecture (MDA) and Executable UML [5], allowing to simulate models, i.e., to provide simulation testing. The pure formal models (e.g., Petri Nets, calculus, etc.) allow to use formal or simulation approaches to complete the testing, verification, and analysis activities. There is no need of model generation or transformation due to simulation purposes. We only add simulated inputs and expected results and can change any model element for its simulated version [6].

The development methods such as MDA [7] allow for semi-automatic translation of designed models to implementation language (i.e., the code generation). Nevertheless, the result has to be finalized manually, so it entails a possibility of semantic mistakes or imprecision between models and transformed code. In comparison with semi-formal models, formal models bring the clear and understandable modeling and the possibility to check correctness with no need for model transformation. The design technique, which is taken into account in this paper [8] derives benefit from formalisms of Object Oriented Petri Nets (OOPN) [9], [10] and DEVS [11]. These formalisms can be directly interpreted and, consequently, integrated into the target system [12].

The paper is organized as follows. First, we briefly introduce used formalisms of OOPN, DEVS, and UML and the design technique. The next chapter deals with architectural description using different formalisms and the fourth section deals with description of behavior.

II. MODELING FORMALISMS

A. UML

The UML modeling uses a notion *view*. A view of a system is a projection of the system on one of its relevant aspects. Such a projection focuses on certain aspects and ignores others. Therefore it is useful to have different views of a system. UML uses multiple notations (tools) for exhibiting different *views* of the system. We can distinguish four main views: The *structural view* describes layout between objects and classes, their associations and their possible communication channels. The *behavioral view* describes, how the system components interact, and characterizes the response to external system operations. The *data view* describes the state of the system units (objects) as well as their relationships. The *interface view* focuses on the encapsulation of system parts, and the possible usage from outside.

UML currently has as many as eleven different notations, which constitute different views of UML designs. In the following, we briefly discuss several notations and their usage to describe certain aspects. *Use case diagrams* display the relationship among actors and use cases. A use case is a set of scenarios describing an interaction between a user and a system. The two main components of a use case diagram are use cases and actors. Use case diagrams deal with an interface and behavioral view at the border of the system. *Sequence diagrams* can be used to demonstrate the interaction of objects in a use case. They generally show the sequence of events that occur. Sequence diagrams therefore clearly define behavioral aspects but are based on structural and interface views. They do not describe an internal behavioral of objects. *Class diagrams* are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. They are the central notion for structural aspects. *State Diagrams* are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and tracks the different states of its objects through the system. *Activity Diagrams* are defined as a special case of a state diagrams. They could be useful for describing internal processing of operations or use cases. Activity diagram models a dynamic flow controlled by internal stimuli.

B. OOPN

An OOPN is a set of classes specified by high-level Petri nets. An *object-oriented Petri net* is a triple (Σ, c_0, oid_0) where Σ is a system of classes, c_0 an initial class, and oid_0 the name of an initial object from c_0. A *class* is mainly specified by an object net and a set of method nets. Object nets describe possible autonomous activities of objects, while method nets describe reactions of objects to messages sent to them from the outside.

Object nets consist of places and transitions. Every place has its initial marking. Every transition has conditions (i.e., inscribed testing arcs), preconditions (i.e., inscribed input arcs), a guard, an action, and postconditions (i.e., inscribed output arcs). *Method nets* are similar to object nets but, in addition, each of them has a set of parameter places and a return place. Method nets can access places of the appropriate object nets in order to allow running methods to modify states of objects, which they are running in. *Synchronous ports* are special (virtual) transitions, which cannot fire alone but only dynamically fused to some other transitions, which activate them from their guards via message sending. Every synchronous port embodies a set of conditions, preconditions, and postconditions over places of the appropriate object net, and further a guard, and a set of parameters. Parameters of an activated port s can be bound to constants or unified with variables defined on the level of the transition or port that activated the port s. *Negative predicates* are special variants of synchronous ports. Its semantics is inverted—the calling transition is fireable if the negative predicate is not fireable.

The OOPN dynamics is based on high-level Petri net dynamics, but the semantics of a transition is little bit modified. A transition is *fireable* for some binding of variables, which are present in the arc expressions of its input arcs and in its guard expression, if there are enough tokens in the input places with respect to the values of input arc expressions and if the guard expression for the given binding evaluates to true. A state of the running OOPN model has the form of a marking of a system of net instances. Each net marking consists of places and transitions marking.

C. DEVS

DEVS is a formalism, which can represent any system whose input/output behavior can be described as sequence of events. DEVS is specified as a structure

$$M = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta)$$

where X is the set of input event values, S is the set of state values, Y is the set of output event values, δ_{int} is the internal transition function, δ_{ext} is the external transition function, λ is the output function, and ta is the time advance function. At any time, the system is in some state $s \in S$. If no external event occurs, the system is staying in state s for $ta(s)$ time. If elapsed time e reaches $ta(s)$, then the value of $\lambda(s)$ is propagated to the output and the system state changes to $\delta_{int}(s)$. If an external event $x \in X$ occurs on the input in time $e \leq ta(s)$, then the system changes its state to $\delta_{ext}(s, e, x)$.

This way we can describe atomic models. Atomic models can be coupled together to form a coupled model. The later model can itself be employed as a component of larger model. This way the DEVS formalism brings a hierarchical component architecture.

D. DEVS and OOPN Wrapping

The DEVS formalism, especially its composite model concept, is suitable as a component platform for multi-paradigm modeling and simulation where atomic models are specified by other formalisms. On the other hand, OOPN is a powerful language allowing a high-level description of model dynamics. The OOPN formalism can be wrapped in the DEVS formalism. In such a case, the OOPN models are atomic components of a hierarchical DEVS model.

Let $M_{PN} = (M, \Pi)$ be a DEVS component M, which wraps an OOPN model Π, c_0 is an initial class of the model Π, and oid_0 is an initial object of the class c_0. Then we define a set of places of the object net oid_0 as $P(oid_0)$, a set of input places $P_{inp} \subseteq P(oid_0)$, and a set of output places $P_{out} \subseteq P(oid_0)$, where $P_{inp} \cap P_{out} = \emptyset$.

Sets X, Y from DEVS formalism are to be specified as structured sets. It allows to use multiple variables for specification of state and we can use input (V_X) and output ports (V_Y) for input and output events specification. Then we can define a mapping of OOPN places into DEVS ports as bijections $map_{inp} : P_{inp} \rightarrow V_X$ and $map_{out} : P_{out} \rightarrow V_Y$.

Informally, if an OOPN model is defined as a DEVS component, then an object net of initial class defines input and output places, this class is instantiated immediately the component is created, and the defined places serve as input or output ports of the component.

E. Design Techniques

As in every design techniques the most problem at its usage is by estimation to abstraction level. Primarily there have to be found essential objects of a modeled system and their relationships. There we can successfully employ resources of UML such us *Use Case*, *Sequence*, *Class* diagrams. As UML, the development processes based on OOPN use the concept of *view*. The basic view is the *data view*, the structure encapsulating data and basic behavior on them. The data view can have different roles in the system, whereas each role is described by another view, which encapsulates the original view. Therefore, views create a hierarchical structure where the higher view encapsulates the view on the lower level. The communication and synchronization between views are provided by means of synchronous ports and negative predicates (more details will be demonstrated in the chapter IV-B).

III. Specification of System Architecture

The system design should be structured into units, whereas each unit is responsible to serve relatively independent activities. The unit can be an object, a package, a component, etc. These units communicate each other by means of specified interfaces. The interface is usually formed by the protocol offered by the object, by the class of component, etc. The units communicate by message passing whereas this communication is usually synchronous.

A. UML-based Specification

Let us have the units U_1 and U_2, the unit U_1 has an interface which is specified by the class C_1^{U1}. If the object from unit U_2 wants to use the unit U_1, it sends a message M_1 to an object O_1^{C1}, which is derived from the class C_1^{U1}. It means that there has to be instances of classes, which constitute the unit interface. If there is a special request to interface (just one class instance, asynchronous communication, etc.), it should be handled in a special way. The system architecture is obviously specified by class diagrams and package diagrams in UML language. The example of this situation is shown in the Figure 1.

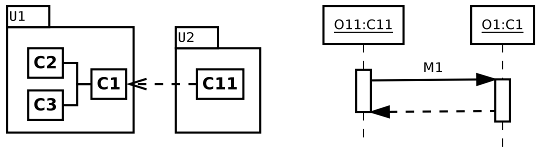

Figure 1. The architectural design using UML.

B. OOPN and DEVS-based Specification

Now, let us have the similar situation, but we use the formalisms DEVS and OOPN for specification of the system architecture. This approach considers the unit as a component based on the DEVS formalism. It means that the interface is established by ports, components are connected via their input and output ports. Let us have the unit U_1 having one input port P_{I1}^{U1} and the unit U_2 having one output port P_{O1}^{U2}. If the component U_2 wants to communicate with the component U_1, it puts the data into the port P_{O1}^{U2}. The data are then transferred to the port P_{I1}^{U1} and the component can react. This communication is asynchronous (the sender does not wait for the answer). This situation is shown in the Figure 2a), which depicts the connection between two units U_1 and U_2 and the Figure 2b), which depicts implementation in the initial object of the unit U_2. There is the place `PO1` representing the output port P_{O1}^{U2}. After the data is put in this port, the next operation of the component U_2 represented by the transition `t2` can execute.

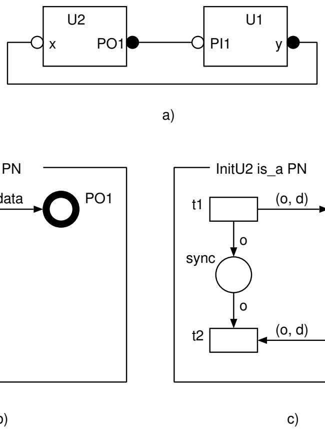

Figure 2. The architectural design using DEVS and OOPN.

Of course, the different requests to interface (e.g., synchronous communication) should be handled, but it can be modeled by a simple way as it is shown in the Figure 2c). The specific identification (object, symbol, etc.) is joined to transferred data and the next operation represented by the transition `t2` will be executable after the respond is accepted, i.e., the data joined with the identification is put in the input port x (place `x`).

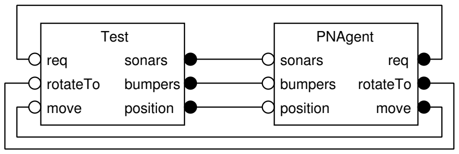

Figure 3. The architectural design using DEVS and OOPN: real example.

The Figure 3 shows a real example of using DEVS and OOPN. There are two units (components) representing one agent (the component `PNAgent`) and its simulated environment (the component `Test`). `PNAgent` receives input from the environment (actual data from sensors) and react by putting commands into its output ports.

IV. Specification of System Behavior

There are a number of techniques in UML to model dynamic aspects of a system. If these techniques are used, the designer statically describes a behavior of a system in a design phase and he cannot make certain of his partial ideas about the system behavior. Next, there can be a problem with understanding to models (diagrams), which arise from graphical whatness of these notions. We could describe this situation by following words: *The nice thing about graphical description techniques is that everyone understands them, the bad thing is that everyone understands them in a different way.* Therefore, the given notions lack a formal foundation, which would make possible understand the model in a only way and also make possible analysis and verifications of the system.

For demonstration purposes, we have chosen a part of the PNtalk system. PNtalk is the tool intended to model a simulate systems using OOPN. We will model a PNtalk processor, which is a central part of the PNtalk system and execute an OOPN model. We will keep an experimental implementation from 2008 [13] in view. First, we use UML to model a chosen part. Second, we adapt these models into OOPN-based models to show how they can be used together.

A. UML-based Modeling

The PNtalk model specifies an OOPN. The OOPN is a set of classes of objects; objects are instances of classes. These classes are translated into *an internal representation* of the PNtalk processor. An internal representation consists of objects corresponding to classes defined by a model. These objects are derived from special classes `PNClass` and `PNSuperClass`, as we can see in the Figure 4.

There we can see that both classes and objects of OOPN are represented by instances of PNtalk classes `PNClass`

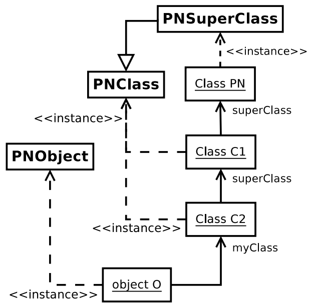

Figure 4. Class diagram of the PNtalk core.

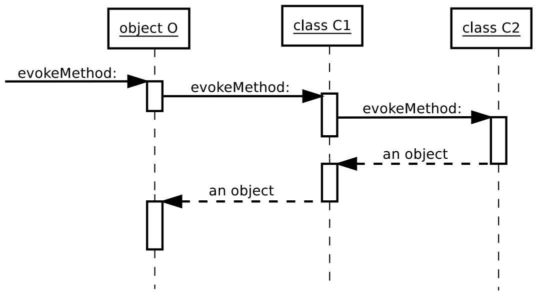

Figure 5. Sequence diagram of the method invocation.

(or `PNSuperClass`) and `PNObject`. Objects whose name starts with *Class* represent classes of an OOPN model. An object named *object O* represents an instance of OOPN class *C2* during a simulation of OOPN model. An OOPN class *Class PN*, derived from class `PNSuperClass`, is a special object, which always stays at the top of an inheritance hierarchy of OOPN classes. The inheritance of OOPN classes are represented as an association between appropriate objects, instances of the class `PNClass` (respectively `PNSuperClass`). The object of the OOPN super class plays the role of *superClass*. The relationship between OOPN object and its OOPN class is represented by an association between appropriate object. The object of OOPN class plays the role of *myClass*. Summary, the Figure 4 shows one object (it is derived from a class `PNObject`), two OOPN classes (they are derived from a class `PNClass`) and one OOPN class, which is derived from class `PNSuperClass`.

We depict functionality by a part of the PNtalk processor—calling methods of OOPN objects. Because classes of the OOPN model are translated into internal-representation objects in implementation environment, there cannot be applied natural inheritance of given environment there. On that account there must be implemented own inheritance hierarchy direction. Let us specify a letter `O` as an OOPN object, which is derived from a OOPN class `C`. Remember these two elements are objects in the PNtalk processor internal-representation. Now, if we attempt to call an OOPN method M of an object O, the object O devolve this requirement upon a class C. The class C looks for an object, which represents a called method M. If this object has been found, the class C gives it back; if does not, the class C generates a fault. This have been a simple variant without an inheritance. As soon as we add an inheritance, the scenario of an unsuccessful search is changed. The class attempt devolve a requirement upon a superior class (if exists), which process this requirement in the same way like there have been already said. Only if a top class in inheritance hierarchy does not find a called method, there is generated a fault.

If we take a look at the Figure 5, we can see a sequence diagram of an object behavior based on external events. A filled arrow-head stands for an invocated event, the dashed one stand for a returned result of event processing.

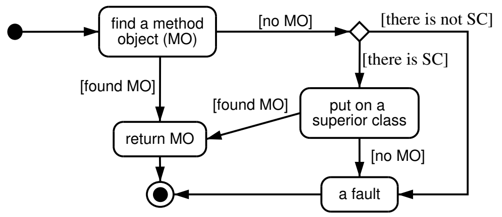

Figure 6. Activity diagram of the method invocation.

Now we focus on an internal behavior of an object representing a PNtalk class close to a method invocation. This is depicted at the Figure 6 subscribing with a description, which has been already depicted there. The letters SC at the picture means a shortcut of *a super class*.

B. OOPN-based Modeling

We have dealt with UML modeling of our chosen part of a system so far. Now we pay attention to OOPN-based modeling. We create the OOPN model according to previous diagrams and designed methodology [8].

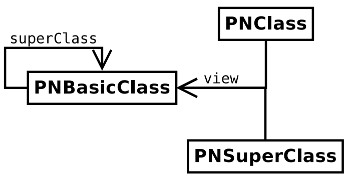

Figure 7. Class diagram of the PNtalk core in the SBD methodology.

The model consists of three classes (see the Figure 7)—PNBasicClass representing a data view, PNClass representing a view of the OOPN class, and PNSuperClass representing the first class in the OOPN inheritance hierarchy. The first class has a bit different behavior, hence it is modeled as a special case. The views modeled by classes PNClass and PNSuperClass represent roles what the basic subject can play in the system. We can see, that the inheritance in design is replaced by object composition, each new view creates a new composition of objects.

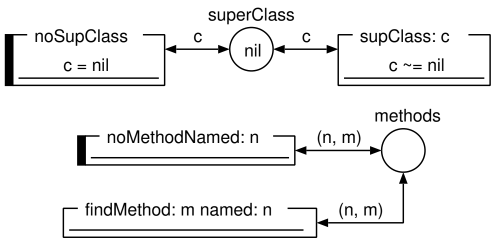

Figure 8. OOPN model representing the class PNBasicClass.

The Figure 8 shows an object net of the class PNBasicClass representing a view of data. There are modeled two places, which store information about the super class (the place superClass) and defined methods (the place methods). The super class is always represented by some of defined view—in this examples it can be view derived from the class PNClass or PNSuperClass. There are defined a pair of synchronous port and negative predicate for each place. Synchronous ports (supClass: and findMethod:named:) allows for finding and getting appropriate data, i.e., the super class or method with given name. If such a method exists (is stored in the place methods), the object representing such a method is bound to the variable m and the calling transition is able to work with it (see the Figure 9). Similarly, if the super class is not equal to nil and the synchronous port is called, the super class is bound to the variable c. The negative predicates (noSupClass and noMethodNamed:) are true if there is no super class or no method with given name. Of course, there should be methods for setting the super class and for adding methods, but they are not shown due to simplicity.

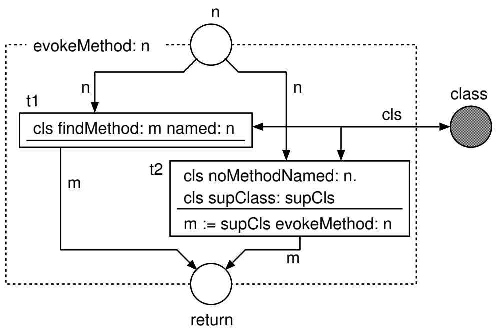

Figure 9. Method evokeMethod: of the view (class) PNClass.

The Figure 9 shows the method evokeMethod: of the view PNClass, which encapsulates the class PNBasicClass (the encapsulated object is stored in the place class). The method has one parameter (a name of the method, which is to evoked). This method process a requirement to a method invocation of OOPN object (and its relevant OOPN class).

If any OOPN method is invocated, the method *evokeMethod:* of the relevant view `PNClass` is called. The method is processed as follows. The transitions `t1` and `t2` are tested. The system of pairs of synchronous port and negative predicate ensures that just one of these transition can be fired. If the encapsulated object of the class `PNBasicClass` contains a method matching given name `n`, the synchronous port `findMethod:named:`, together with the transition `t1`, is fired and the found method (the object representing the method, respectively) is bound to the variable `m` and returned as a result of the method.

If there is no such a method, the negative predicate `noMethodNamed:` called from the transition `t2` is true. After firing this transition, the synchronous port `supClass:`, placed as a second condition in the guard, binds the super class to the variable `cls`. Then the method `evokeMethod:` is called on the super class and its result is returned.

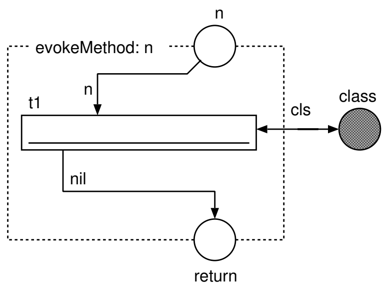

Figure 10. Method `evokeMethod:` of the view (class) PNSuperClass.

If the view on the super class is derived from the class *PNSuperClass*, the search of a method object is unsuccessful and its method *evokeMethod:* give back an object `nil` (see the Figure 10). There should be only one view derived from this class representing the first class *PN* in the OOPN inheritance hierarchy.

V. Conclusion

The paper dealt with formalisms of OOPN, DEVS, and UML used in the system design. In comparison with UML, using formalisms of DEVS and OOPN for the architectural specification decreases a number of communication points and makes the architectural specification well-arranged. Moreover, selected models in UML can be transformed to DEVS component described by OOPN formalisms in a simple way. The presented approach is a part of the development methodology, which allows to use formal models in all phases of system development including as basic design, analysis and also programming means with a vision to allow to combine simulated and real components and to deploy models as the target system with no code generation. In the future, we plan to formalize outlined approach and to present complex case study.

Acknowledgment: This work was partially supported by the BUT FIT grant FIT-S-11-1 and the Ministry of Education, Youth and Sports under the contract MSM 0021630528.

References

[1] J. Arlow and I. Neustadt, *UML and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Professional, 2001.

[2] S. Beydeda, M. Book, and V. Gruhn, *Model-Driven Software Development*. Springer-Verlag, 2005.

[3] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[4] M. Broy, J. Gruenbauer, D. Harel, and T. Hoare, Eds., *Engineering Theories of Software Intensive Systems: Proceedings of the NATO Advanced Study Institute*. Kluwer Academic Publishers, 2005.

[5] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie, *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[6] R. Kočí and V. Janoušek, "OOPN and DEVS Formalisms for System Specification and Analysis," in *The Fifth International Conference on Software Engineering Advances*. IEEE Computer Society, 2010, pp. 305–310.

[7] D. S. Frankel, *Model Driven Architecture: Applying Mda to Enterprise Computing*, ser. 17 (MS-17). John Wiley & Sons, 2003.

[8] R. Kočí and V. Janoušek, "System Design with Object Oriented Petri Nets Formalism," in *The Third International Conference on Software Engineering Advances Proceedings ICSEA 2008*. IEEE Computer Society, 2008, pp. 421–426.

[9] M. Češka, V. Janoušek, and T. Vojnar, *PNtalk — a Computerized Tool for Object Oriented Petri Nets Modelling*, ser. Lecture Notes in Computer Science. Springer Verlag, 1997, vol. 1333, pp. 591–610.

[10] V. Janoušek and R. Kočí, "PNtalk Project: Current Research Direction," in *Simulation Almanac 2005*. FEL ČVUT, Praha, CZ, 2005.

[11] B. Zeigler, T. Kim, and H. Praehofer, *Theory of Modeling and Simulation*. Academic Press, Inc., London, 2000.

[12] R. Kočí and V. Janoušek, *Simulation Based Design of Control Systems Using DEVS and Petri Nets*, ser. Lecture Notes in Computer Science. Springer Verlag, 2009, vol. 5717, pp. 849–856.

[13] V. Janoušek and R. Kočí, "Embedding Object-Oriented Petri Nets into a DEVS-based Simulation Framework," in *Proceedings of the 16th International Conference on System Science*, ser. volume 1, 2007, pp. 386–395.