Object Oriented Petri Nets in Software Development and Deployment

Radek Kočí and Vladimír Janoušek

Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence
Bozetechova 2, 612 66 Brno, Czech Republic
{koci,janousek}@fit.vutbr.cz

***Abstract*—Modeling, implementation, and testing are integral parts of system development process. Models usually serve for description of system architecture and behavior and are automatically or manually transformed into executable models or code in a programming language. Tests can be performed on implemented code or executable models; it depends on used design methodology. Although models can be transformed, the designer has to usually adapt resulted code manually. It can results in inconsistency among design models and their realization and the further development, testing and debugging by means of prime models is impossible. The approach discussed in this paper allows to model and test systems using high-level languages, especially Object Oriented Petri Nets combined with Discrete Event System Specification, whereas models are deployed to the product environment and become integral part of the system.**

Keywords-Object Oriented Petri Nets; DEVS; model deployment.

I. Introduction

Modeling, implementation, and testing are integral parts of system development process. Various models are used in analysis and design phases and usually serve as a system documentation rather than real models of the system under development. The system is then implemented according to these models, whereas the code is either generated from models or is implemented manually. Unfortunately, many implementation differ from designed models because of debugging or system improvement. Consequently, models become out of date and useless.

To solve a problem with manual implementation and impossibility to test designed system using models, the methodologies and approaches commonly known as Model-Driven Software Development are investigated and developed for many years [1], [2] These methods use executable models, e.g., Executable UML [3] in Model Driven Architecture methodology [4], which allows to test systems using models. Models are transformed into another models and, finally, to code. Nevertheless, the resulted code has to often be finalized manually and the problem with semantic mistakes or imprecision between models and transformed code remains unchanged.

The approach to system development, which is presented in the paper, uses formal models as a means for system description as well as system implementation. The basic idea is to have a framework allowing to execute models in different modes, whereas each mode is advisable for another kind of usage—design, testing, and deployment. The system is developed using different kinds of models (from formal models to direct code in a programming language) in simulation, i.e., it is possible to test systems in any state in any time. The design method, which is taken into account in the papers [5][6], does not require model transformations and assumes that models serve for system description as well as system implementation. The formalism of Object-Oriented Petri Nets (OOPN) [7], [8] and Discrete Event System Specification (DEVS) are basic modeling means.

The paper is organized as follows. First, we briefly introduce the used formalisms of OOPN and DEVS in Section III, application framework in Section IV, and design methodology including a simple case study model in Section V. Possibilities to deploy models into product environment will be discussed in Section VI.

II. Related Work

Combination of formal models, simulation, and model deployment is applicable mainly in control software. The use of high-level languages, especially Petri Nets, allows to build and maintain control systems in a quite fast and intuitive way. To control robot application, hierarchical binary Petri nets are used for middleware implementation in a RoboGraph framework [9]. To develop control software for embedded systems, the work which uses Timed Petri Nets for the synthesis of control software by generating C-code [10], the work based on Sequential Function Charts [11], or the work based on the formalism of nets-within-nets (NwN) [12], [13], [14] can be mentioned.

These tools and works allow to *model* systems using a combination of different formalisms, but do not allow to use formal models in system *implementation*. The proposed approach allows to use formal models as a basic design, analysis and programming means combining simulated and real components. The main advantages; there is no need for code generation, and for further investigation of deployed systems, using the same formal models and methods is possible.

III. Used Formalisms

We will briefly introduce the formalisms of Object-Oriented Petri Nets and Discrete Event System Specification in this section.

A. Formalism of Object Oriented Petri Nets

Object orientation of Object-Oriented Petri nets (OOPN) [15] is based on the well-known class-based approach. All objects are instances of classes, every computation is realized by message sending, and variables contain references to objects. This kind of object-orientation is enriched by concurrency. OOPN objects offer reentrant services to other objects and, at the same time, they can perform their own independent activities. The services provided by the objects as well as the autonomous activities of the objects are described by means of high-level Petri nets—services by *method nets*, object activities by *object nets*.

The formalism of OOPN contains important elements allowing for testing object state (*predicates*) and manipulation with object state with no need to instantiate nets (*synchronous ports*). Object state testing can be negative (*negative predicates*) or positive (*synchronous ports*). We can see that synchronous ports can be used for testing as well as for manipulation. *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. *Negative predicates* are special variants of synchronous ports with inverted semantics—the calling transition is fireable if the negative predicate is not fireable.

B. Formalism of DEVS

Discrete Event System Specification (DEVS) [16] is a formalism, which can represent any system whose input/output behavior can be described as sequence of events. The atomic DEVS model is specified as a structure M containing sets of states S, input and output event values X and Y, internal transition function δ_{int}, external transition function δ_{ext}, output function λ, and time advance function ta. These functions describe behavior of the component.

This way we can describe atomic models. Atomic models can be coupled together to form a coupled model CM. The later model can itself be employed as a component of a larger model. This way the DEVS formalism brings a hierarchical component architecture. Sets S, X, Y are obviously specified as structured sets. It allows to use multiple variables for specification of a state; we can use a concept of input and output ports for input and output events specification, as well as for coupling specification. In another words, components are connected by means of ports and event values are carried via these ports.

IV. Application Framework

Since one of the main motivations behind the development of OOPN is a possibility to use Petri nets not only for system modeling but also for system implementation and deployment, we need an application framework, which fulfils two basic requirements. First, to link models and product environment. Second, to work with models in simulations.

A. Interoperability with Product Environment

The models described by means of OOPN can cooperate with objects of the product environment (product objects). Since the framework is implemented in Smalltalk [17], OOPN objects can send messages to Smalltalk objects, and OOPN objects can be directly available in Smalltalk. There are different levels at which the product objects can send messages to OOPN objects—*domain*, *predicate*, and *synchronous port* levels. Domain level allows Smalltalk objects to send messages OOPN objects as though they were Smalltalk objects. Predicate level allows to test predicates and port level allows to perform synchronous ports. Each OOPN object offers special meta-protocol allowing to work at presented levels (it will be shown in the text, later on).

Another way on how to connect OOPN models with their product environment is to use component approach based on DEVS formalism. DEVS component can wrap another kind of formalism, so that each such a formalism is interpreted by its simulator and simulators communicate each other by means of a compatible interface. Let $M_{PN} = (M, \Pi, map_{inp}, map_{out})$ be a DEVS component M, which wraps an OOPN model Π. The model Π defines an initial class c_0, which is instantiated immediately the component M_{PN} is created. Functions map_{inp} and map_{out} map ports and places of the object net of the initial class c_0. The mapped places then serve as input or output ports of the component.

B. System in Simulation

The framework offers a protocol for creating and manipulating models and simulations. Models are usually described by formalisms of OOPN or DEVS, but can be implemented in product environment or can interoperate with product environment. The framework allows to execute models in different simulation modes—simulation in *model time*, simulation in *real time*, and simulation in *combined time*.

Each simulation mode is advisable for another kind of usage. *Model time* is intended for basic design, testing, and analysis of system under development and assumes all components are described by formal models. *Combined time* assumes that the system is descibed by formal models as well as implemented in product environment, i.e., selected simulated components are replaced by their real implementation, whereas simulated components work in model time and real components work in real time. This mode allows to experiment with simulation models in real conditions. *Real* *time* assumes that all components (simulated as well as real) work in real time and is intended for hardware/software-in-the-loop simulation and system deployment.

V. System Modeling Using OOPN and DEVS

The system is modeled and simulated in the application framework, which supports formalisms of OOPN and DEVS, so far. This section will demonstrate modeling methodology based on usage of the application framework.

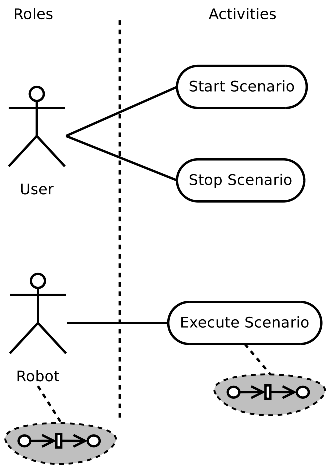

Figure 1. Use Cases of designed system.

A. Modeling Methodology

We will follow the design methodology, which has been presented by Kočí and Janoušek [18]. The modeling process starts with identification of *actors* and *use cases* as a model of *system behavior*. In this phase, the use case diagrams from UML can be used. Based on this diagram, *roles* and their *activity nets* are defined. Roles are based on analysis of actors (actors usually correspond to roles) and activity nets model behavior described by use cases.

Next step is to define an architecture of the system. The architecture can be described by class diagram. Roles and activity nets are encapsulated into classes, furthermore the *subjects* are identified and modeled using classes. Subjects represent information about actors or a group of actors, e.g., one user (a subject) can have more roles (administrator, customer, etc.). The architecture is based on layered modeling of roles and their activities, i.e., each activity encapsulates a role, an activity can encapsulates another activity, etc. Each role and its set of allowed activities (activity nets) can be described by any formalism allowing to define an interface for communication or synchronization, e.g., statecharts, activity diagrams, Petri Nets, etc.

B. System Behavior Modeling

We will demonstrate system modeling and model deployment on a simple case study of a robot control system. First, we identify use cases of the system, as shown in Figure 1. We have found two actors (*User* who can control the system and *Robot* who is controlled) and three use cases (*Execute Scenario* for *Robot*, and *Start Scenario* and *Stop Scenario* for *User*).

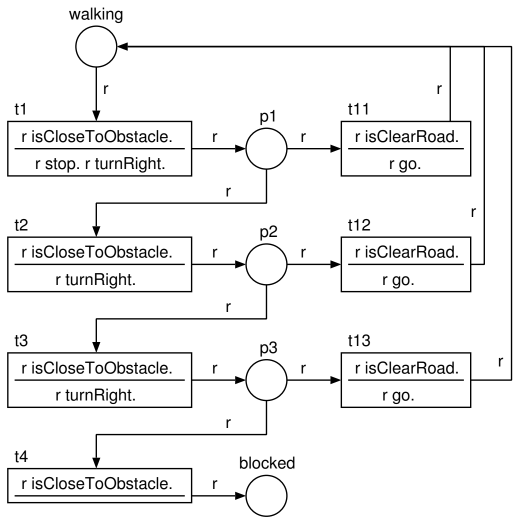

Figure 2. Activity Net *Scenario*.

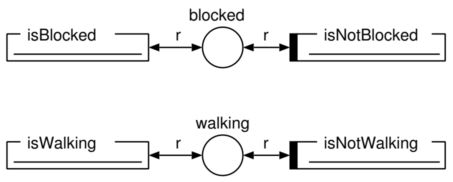

Figure 3. Activity Net *Scenario* – predicates.

Actors represent *roles* and use cases represent *activities* in the system. We aim at the actor *Robot* and the use case *Execute Scenario* in our study. The use case models an activity of the robot. We will suppose very simple activity, which can be described in following algorithm: (1) the robot is walking, (2) if the robot comes upon to an obstacle, it stops, turns to right and tries to walk, (3) if the robot turns three times with no possibility to walk, it stops. The activity net *Scenario* describing the presented behavior of use case *Execute Scenario* is shown in Figure 2.

The robot can be in two stable states—walking or blocked (there is no possibility to walk). Each such a state is represented by appropriate place, i.e., places `walking` and `blocked`. We have to be able to test activity states, therefore the predicates are generated for each such a place—the synchronous port `isBlocked` and the negative predicate `isNotBlocked` for the state `blocked` and similar predicates for the state `walking`. Test predicates are shown in Figure 3.

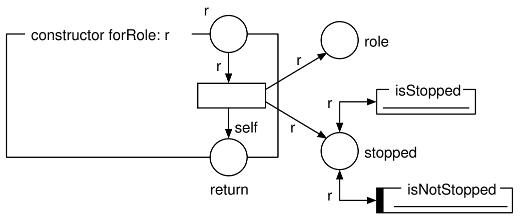

Figure 4. Activity Net *Scenario* – the constructor.

The activity has to be linked to a role in the system—this role is stored in places and serves even as a state token. The role supplies an information about the robot and allows to send commands. Each activity is instantiated for just one role, so that the role is initialized by means of constructor as shown in Figure 4. The new state `stopped` is added—it represents a situation when the robot is stopped but does not stay before any obstacle (e.g., the robot was stopped by user or the activity is being created).

Now, we have to add last element, a possibility to start activity—it is a part of use case *Start Scenario* modeled by method net `start` (see Figure 5), which decides what has to be done based on the activity state. If the activity is *walking*, the method does nothing. If the activity is stopped or blocked, it starts the robot's walk (send a message `go` and moves the token to the place `walking`.

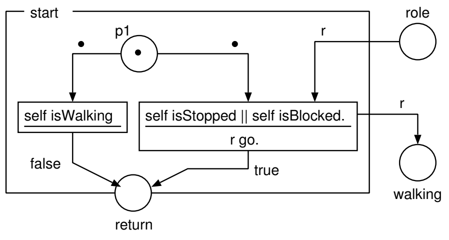

Figure 5. Activity Net *Scenario* – a method net *start*.

C. Architecture Modeling

Each role needs to have its subject, i.e., the object defining information about a subject, which can have different roles in the system. The subject is usually modeled as an object containing efficient data directly or as an interface to database, another system or remote object. The way how to model subjects influences the system architecture.

Figure 6 shows the classes of basic architecture of our example with appropriate stereotypes *Activity Net*, *Role*, and *Subject*. The architecture consists of the subject *RobotDevice*, its role *Robot* and its activity *Scenario*, that have been modeled by OOPN (see the stereotype *PN*). *RobotDevice* represents an interface to the simulated robot and *Robot* represents a role which the robot has in the system. Each method is labeled with one of stereotypes `C` (constructor),

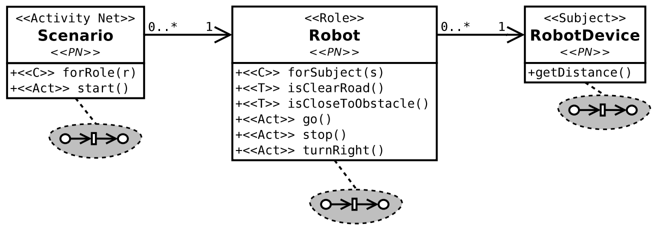

Figure 6. Basic architecture of the case study.

`Act` (activity), and `T` (testing) determining a realization of methods in OOPN (see [19]).

D. DEVS Architecture Modeling

The DEVS architecture of presented case study contains the components *Behavior* and *Subject* as shown in Figure 7. The component *Behavior* describes the system behavior as presented in previous case and the component *Subject* describes a subject of behavior. Subcomponents of the component *Subject* can be modeled by OOPN, programming language, or any other supported formalism. Components are connected via ports *request* and *answer*. The DEVS subcomponent *RobotDevice* is an atomic component, which gets a request string at its input port $request$, asks a robot for answer, and puts this answer to its output port $answer$. This architecture allows to exchange components in a very simple way, because components are connected only by means of ports.

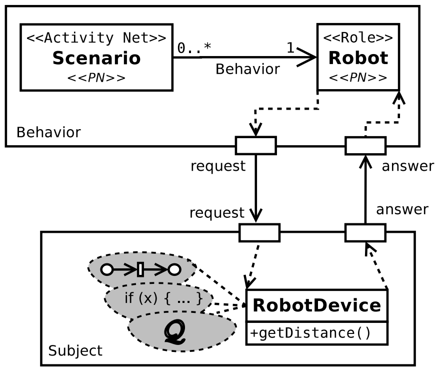

Figure 7. DEVS architecture of the case study.

VI. Software Deployment with Models

This section will demonstrate possibilities of keeping models in the deployed system. It is based on the application framework allowing to interoperability of models and product environment.

A. Implementation with Basic Architecture

A possible model of the role *Robot*, which is based on architecture described in Figure 6, is shown in Figure 8. The role checks actual distance of robot to the obstacle each 10 time units and offers information about robot's position by means of predicates *isClearRoad* and *isCloseToObstacle*. To get information about the distance, the role asks its subject by sending a message *getDistance*.

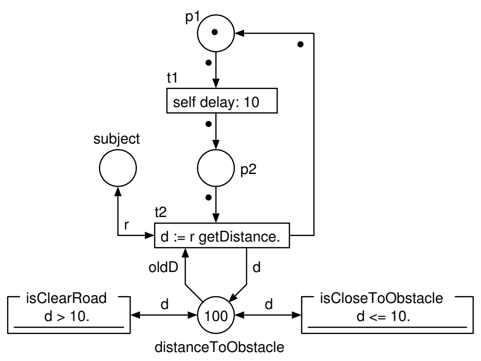

Figure 8. The role *Robot* – implementation for basic architecture.

We can exchange the simulated subject by real interface to the controlled robot. It is very simple—we only create instances of appropriate classes and do not care about used formalism. Figure 9 of Smalltalk code shows creating a subject as an instance of a Smalltalk class. This subject cooperates with a role and an activity modeled by OOPN. The object *Repos* represents the storage of all classes and simulations using OOPN or DEVS formalisms.

```
cAct  := Repos componentNamed: 'Scenario'.
cRole := Repos componentNamed: 'Robot'.
subj := RobotDevice new.
role := cRole forSubject: subjR.
actS := cAct forRole: roleR.
```

Figure 9. Accessing OOPN objects from Smalltalk.

Now, we demonstrate an accessing OOPN objects from product environment of Smalltalk. We send a command to start walking by means of a message `go`—the message passing is provided in the standard form. To test an object state, the predicates should be used. Since they are not ordinary methods, we have to access them in a special way. First, we obtain a special meta-protocol by sending a message `asPredicate`. Second, we can call synchronous port or negative predicate in the standard form of message passing. Third, the result represents a state of a called port/predicate, which has been tested. In our example, we test the predicate `isCloseToObstacle` and if the result is true, then we stop robot's walking by sending a message `stop`. The example is shown in Figure 10.

```
role go.
r := role asPredicate isCloseToObstacle.
r ifTrue: [ role stop ].
```

Figure 10. Message passing and predicate testing.

Of course, proposed solution is not sufficient for our case, because we need to test this condition until it becomes true. Therefore we can use one of following ways—to use waiting for specified condition or to define *a listener*. The first way is shown in Figure 11. We simply use a message `waitFor:` from the meta-protocol, which blocks until the specified condition becomes true, i.e., the port `isCloseToObstacle` becomes fireable.

```
role go.
role asPredicate waitFor: #isCloseToObstacle.
role stop.
```

Figure 11. Waiting for a condition.

Second way is shown in Figure 12. It uses a message `listener:for:` from meta-protocol to define a listener, which is activated if the condition becomes true, i.e., the port becomes fireable.

```
role go.
role asPredicate
      listener: self
      for: #isCloseToObstacle.
```

Figure 12. Setting a listener.

The activation of listener means that the special message `conditionSatisfied:` is sent to object, which is specified as a first argument. The example of its implementation is shown in Figure 13.

```
method conditionSatisfied: aCond
    (aCond == #isCloseToObstacle)
        ifTrue: [ role stop ].
```

Figure 13. Listener implementation.

B. Implementation with DEVS Architecture

Because the architecture changes, we have to modify classes describing system behavior. The component *Behavior* encapsulate OOPN model, which defines the class *Robot* as its initial class, so that ports are mapped to places of the *Robot* object net. This modified object net is shown in Figure 15. Place named *request*, resp. *answer*, corresponds to output port *request*, resp. input port *answer*.

The example of accessing DEVS components and their object interface is shown in Figure 14. First, we get a DEVS simulation named *R01*, which is based on architecture from Figure 7. Second, we obtain DEVS component *Behavior*, which is able to communicate through its ports. Nevertheless, this component is described by OOPN, so that it is possible to use object interface of its initial object (an instance of the class *Robot*) too. To get the object interface, we send a special message `objectInterface` from the component meta-object protocol.

```
s1 := Repos componentNamed: 'R01'.
cB := c1 componentNamed: 'Behavior'.
iB := cB objectInterface.
```

Figure 14. Obtaining object interface to the inital object.

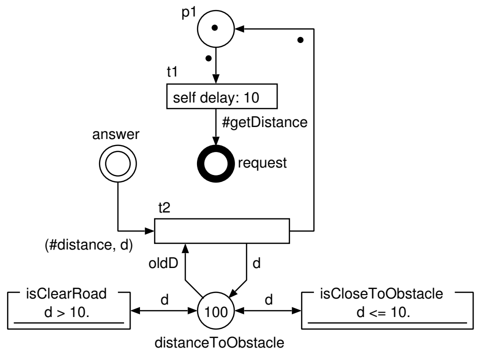

Figure 15. The role *Robot* – implementation for DEVS architecture.

VII. Conclusion and Future Work

The paper dealt with a possibility to deploy formal models to target application using specific application framework. It allows to use formal models as a basic design, analysis and programming means combining simulated and real components. The main advantage of that approach is no need for code generation and further investigation of deployed systems using the same formal models.

The proposed approach has one main disadvantage—usage of application framework, which interprets formal models directly demands of increased requirements on memory size and system performance. The future research will aim at efficient representation of choosed formal models and interoperability with another product environment. The application framework will be adapted to new conditions having lesser requirement for resources.

Acknowledgment

This work has been supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by BUT FIT grant FIT-S-11-1, and by the Ministry of Education, Youth and Sports under the contract MSM 0021630528.

References

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

[2] 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.

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

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

[5] 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.

[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] 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.

[8] 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.

[9] J. L. Fernandez, R. Sanz, E. Paz, and C. Alonso, "Using hierarchical binary Petri nets to build robust mobile robot applications: RoboGraph," in *IEEE International Conference on Robotics and Automation*, 2008, pp. 1372–1377.

[10] C. Rust, F. Stappert, and R. Kunnemeyer, "From Timed Petri Nets to Interrupt-Driven Embedded Control Software," in *International Conference on Computer, Communication and Control Technologies (CCCT 2003)*, 2003.

[11] O. Bayo-Puxan, J. Rafecas-Sabate, O. Gomis-Bellmunt, and J. Bergas-Jane, "A GRAFCET-compiler methodology for C-programmed microcontrollers, In Assembly Automation," *Assembly Automation*, vol. 28, no. 1, pp. 55–60, 2008.

[12] R. Valk, "Petri Nets as Token Objects: An Introduction to Elementary Object Nets." in *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, vol. 120. Springer-Verlag, 1998.

[13] D. Moldt, "OOA and Petri Nets for System Specification," in *Object-Oriented Programming and Models of Concurrency*. Italy, 1995.

[14] L. Cabac, M. Duvigneau, D. Moldt, and H. Rölke, "Modeling dynamic architectures using nets-within-nets," in *Applications and Theory of Petri Nets 2005. 26th International Conference, ICATPN 2005, Miami, USA*, 2005, pp. 148–167.

[15] V. Janoušek and R. Kočí, "PNtalk: Concurrent Language with MOP," in *Proceedings of the CS&P'2003 Workshop*. Warsaw University, Warsawa, PL, 2003.

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

[17] A. GoldBerk and D. Robson, *Smalltalk 80: The Language*. Addison-Wesley, 1989.

[18] R. Kočí and V. Janoušek, "Modeling and Simulation-Based Design Using Object-Oriented Petri Nets: A Case Study," in *Proceeding of the International Workshop on Petri Nets and Software Engineering 2012*, vol. 851. CEUR, 2012, pp. 253–266.

[19] R. Kočí and V. Janoušek, "Specification of UML Classes by Object Oriented Petri Nets," in *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*. Xpert Publishing Services, 2012, pp. 361–366.