Formal Models in Software Development and Deployment: A Case Study

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@fit.vutbr.cz
†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 result in an inconsistency among design models and their realization and the further development, testing and debugging by means of prime models is impossible. This work summarizes the design methodology based on the formalism of Object Oriented Petri Nets combined with Discrete Event System Specification and demonstrates its usage in the system development and deployment on the simple robotic system case study. The goal is to use the same formalisms for system modeling as well as for system implementation, so that to keep designed models in the deployed system.**

Keywords–Object Oriented Petri Nets, Discrete Event System Specification, multi-paradigm modeling, model deployment.

I. Introduction

This work is based on the paper [1], which is extended of detailed explanation of the design methodology and its usage for system development and deployment. It is demonstrated on simple, but fully described, case study.

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, implementations often differ from the 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 Consequently, models become out of date and useless.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 [2], [3] These methods use executable models, e.g., Executable UML [4] in Model Driven Architecture methodology [5], 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 [6], [7], 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) [8], [9] and Discrete Event System Specification (DEVS) are basic modeling means.

The paper is organized as follows. First, we will attend to related work in Section II. The formalism of OOPN will be briefly introduced in Section III. Basic principles of modeling methodology will be described in Section IV, different modeling means will be compared in Section V, the approach to model system behavior will be presented in Section VI, and Section VII pays an attention to the architecture modeling. Finally, possibilities to deploy models into product environment will be discussed in Section VIII and Section IX concludes the paper and describes a future work.

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 [10]. To develop control software for embedded systems, the work that uses Timed Petri Nets for the synthesis of control software by generating C-code [11], the work based on Sequential Function Charts [12], or the work based on the formalism of nets-within-nets (NwN) [13], [14], [15] 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. Formalism of Object Oriented Petri Nets

We will briefly introduce the formalisms of Object-Oriented Petri Nets. Object orientation of Object-Oriented Petri nets (OOPN) [16] 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*).

An example illustrating the important elements of the OOPN formalism is shown in Figure 1. There are depicted two classes `C0` and `C1`. The object net of the class `C0` consists of places `p1` and `p2` and one transition `t1`. The object net of the class `C1` is empty. The class `C0` has a method `init:`, a synchronous port `get:`, and a negative predicate `empty`. The class `C1` has a method `doFor:`.

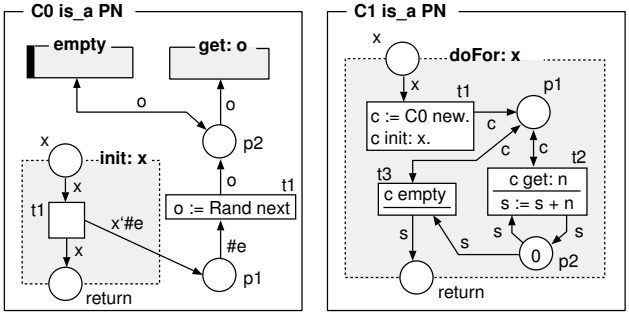

Figure 1. An OOPN example.

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.

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 $sync$ can be bound to constants or unified with variables defined on the level of the transition or port that activated the port $sync$. An example is shown in Figure 1—the port `get:` (class `C0`) having one formal parameter `o` is called from the transition `t2` (class `C1`) with free variable `n`—it means that the variable `n` will be unified with the content of the place `p2` (class `C0`).

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 passed variable cannot be unbound (the unification is impossible) and the predicate cannot have a side effect. An example is shown in Figure 1, the predicate `empty` (class `C0`). This predicate is called from the transition `t3` (class `C1`)—it means that the transition `t3` will be fireable if the place `p2` (class `C0`) is be empty.

Let us investigate what happens after calling the method `doFor:` with a value `3` on an instance of the class `C1` (the instance will be denoted by obj_{C1}). First, the transition `t1` is fired with following actions: the instance of the class `C0` is created (the instance will be denoted by obj_{C0} and the reference to this object is assigned to the variable `c`) and initialized by the method net `init:`. It puts the symbol `#e` to the place `p1` of the object net obj_{C0} three times. The transition `t1` of the object net obj_{C0} generates three random numbers and puts them into the place `p2`. Second, the transition `t2` of the object net obj_{C1} tests if there is any object (a value) in the object net obj_{C0} by testing the synchronous port `get:`. If its evaluation is true, the transition `t2` is firable. If the transition `t2` fires, the synchronous port `get:` fires too. Since the variable `n` is free, the variable `n` is unified with a random number from the place `p2` of the object net obj_{C0}. The transition `t2` of the object net obj_{C1} then adds this value to the sum (the variable `s`). Otherwise, the the transition `t3` of the object net obj_{C1} tests if there is no value in the object net obj_{C0}—then the negative predicate `empty` is firable. If the transition `t3` fires, it places the sum (the variable `s`) to the return place as a method result. So, an invocation of the method `doFor:` leads to random generation of `x` numbers and to return of their sum.

IV. Modeling Methodology Principles

This section introduces modeling methodology, which has been presented by Kočí and Janoušek [17], and simple case study. Only basic principles of methodology will be shown here; details and the complex model of proposed case study will be being developed in Sections VI and VII.

A. Modeling Process

The modeling process is split up into three basic phases—identification of model elements, modeling the system architecture, and modeling the system behavior. Different modeling means are used in different steps, nevertheless, theses means are linked together. Brief description of basic phases and used modeling means follows:

- *Basic model elements* are *users* of the system, their *roles*, and *activities* of the system. To identify them, the *use case diagram* from UML can be used. Roles

are modeled by means of actors and activities by means of use cases.

- *System architecture* is modeled by means of class diagrams from UML. Modeled classes are linked to elements of use case diagrams as follows:
  - roles are represented by a group of classes modeling logic view at roles behavior,
  - activities are represented by a group of classes modeling functionality of the system,
  - users are represented by a group of classes modeling data of roles and accessing potentially present communication channels.
- *System behavior*. Behavior of roles and activities is modeled by means of Object oriented Petri nets formalism. It can also be modeled by any other formalism allowing to define workflow scenarios and offering an interface for workflows synchronization, e.g., statecharts, activity diagrams, or other kind of Petri nets. The comparison of chosen formalisms is done in Section V.

B. Layered Architecture of the Modeling Process

First, the relationship between *user* and *role* has to be explained. The role defines a work context the user can act in. For instance, the user working with a conference system can act as an author, a reviewer, a chair, or a combination of these roles. Thus, users act through their roles in the system and each user can have more roles. Nevertheless, the system can have other objects than users, that have the same or similar meaning, that is to represent data base, either for some of present roles or for data needed to be stored in the system. We denote such objects by a notion *subjects*.

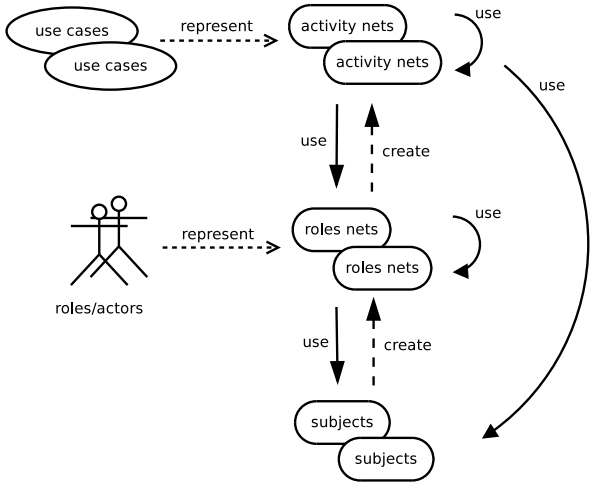

Figure 2. Design Method – Layers in the model.

Figure 2 shows model elements and their relationships in the design process. The design method distiguishes following model elements:

- *Subjects* represent a base for data storage or accessing communication channels.
- *Role nets* are derived from *actors* modeled in use case diagrams. Role nets model logic view at roles in the system and offer the communication protocol to the subject depending on the role intention.
- *Activity nets* are derived from use cases and model system functionality. The model is based on workflow definition and uses roles and subjects.

Model elements are organized in layers. Relationshisps are depicted by arrows—the solid arrow from *sender* to *receiver* represents a relationship a *sender* uses a *receiver*. For instance, activity nets can use roles nets, subjects, but also other activity nets. The dashed arrow from *sender* to *receiver* represents a relationship a *sender* creates a *receiver*. For instance, activity nets are created by role nets.

C. Simple Case Study

We will demonstrate basic principles of the design method on the example (simple case study) of a robot control system. We have a system with robots, where a motion of each robot is controlled by the same scenario, which is described by the following algoritm: (1) the robot is walking; (2) if the robot comes upon to an obstacle, it stops, turns right and tries to walk; (3) if the robot cannot walk, it turns round and tries to walk; (4) if the robot cannot walk, it stops. User can start and stop this scenario anytime.

V. UML and OOPN in the Design Process

As it was mentioned, the different formalisms or means can be used to describe system behavior. The comparison of using chosed UML models and formalism of OOPN will be presented in this section. It will be demonstrated on the previously instroduced case study.

First of all, the designer would identify model elements using use case diagrams. After analysing the case study specification, we can model use cases as shown in Figure 3. There we can find an actor *User* and a use case *Execute Scenario* representing the control algorithm.

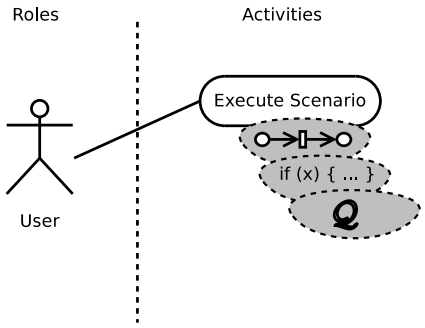

Figure 3. First Use case diagram of designed system.

A. UML in the Design Method

In UML, the use case specification is usually described informally by means of pure text or semi-structured text displayed in a table. Figure 4 shows such a table, which specify a behavior of the use case *Execute Scenario*. The table describes an algorithm in an informal way using keywords for text structuring, e.g., *IF–ELSE* branching, *REPEAT* a step, etc.

Name	Execute Scenario
Sequence of steps	
1. The scenario starts by user's stimulus 2. IF there is a clear road 2.1. the robot goes straight 2.1. repeat step 2 3. ELSE 3.1. the robot turns right 3.2. IF there is not clear road 3.2.1. the robot turns round 3.2.2. IF there is a clear road 3.2.2.1. REPEAT step 2	
Alternative sequence of steps	
1. User can stop this scenario anytime	

Figure 4. Specification of the Use Case *Execute Scenario*.

Another way to describe use case specifications is to use digrams from UML such as *activity diagram* or *statecharts*. Their usage allows for more precise description of behavior, which is based on predefined elements with clear semantics.

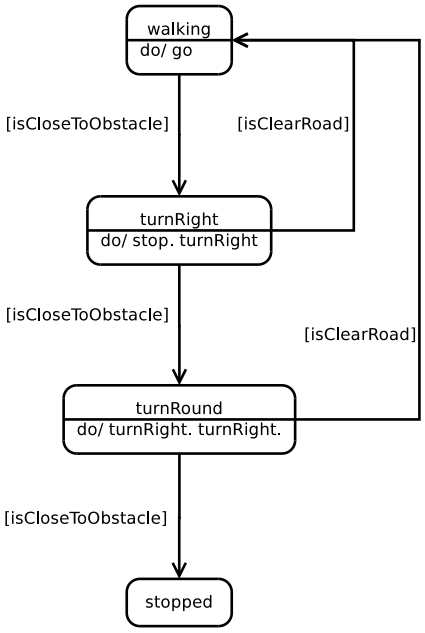

Figure 5. Statechart of the use case *Execute Scenario*.

The statechart modeling the use case *Execute Scenario* is shown in Figure 5. Four different states of modeled activity have been identified:

- *walking* means the robot walks
- *turnRight* means the robot turns right for the first time it came at an obstacle
- *turnRound* means the robot turns round for the second time it cannot go on
- *stopped* means the robot cannot go on

Each state containts commands that are executed in the state—*go* instructs the robot to go straight, *stop* instructs the robot to stop, and *turnRight* instructs the robot to turn right. If the command *turnRight* is sent twice, it means that the robot turns round.

Transitions between states are modeled by means of arcs whereas each transition can be conditioned. In the example, there are two conditions for transitions:

- *isCloseToObstacle* means the robot came at an obstacle and cannot go on
- *isClearRoad* means the robot can go on

B. Object Oriented Petri Nets in the Design Method

Methods that have been presented in Section V-A allow for use case description but their validation can be problematic because of impossibility to check models either by formal means or by simulation. Of course, there are tools and methods [4], [5] that allow to simulate modified UML diagrams. Nevertheless, there is still a strict border between *design* and *implementation*. On the other hand, if the formalisms allowing *to design* as well as *to implement* the system is used, we needn't care about borders and problems that can arise during a transition from design to implementation, and vice versa. One of such formalisms, which can be used to model use case (activity) behavior is the formalism of Object-Oriented Petri Nets (OOPN).

Let us continue in the example of *Execute Scenario*. The activity net *ExecuteScenario* of the use case *Execute Scenario*, which is described using OOPN, is shown in Figure 6.

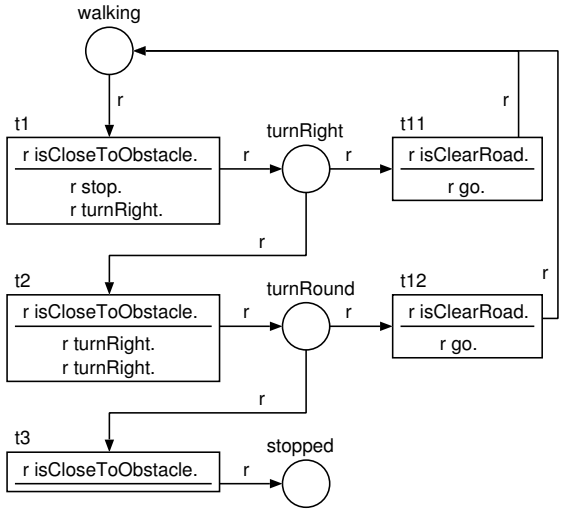

Figure 6. Activity Net *ExecuteScenario*.

The activity net contains elements similar to the statechart model shown in Figure 5. States are modeled by places `walking`, `turnRight`, `turnRound`, and `stopped` with the same meaning. Nevertheless, states `turnRight` and `turnRound` are only temporal and the activity goes through these ones to the one of stable states `walking` or `stopped`.

The control algorithm is represented by a sequence of transitions whereas each transition is conditioned by an event representing a change on robot's state. For instance, the transition `t1` has a condition *isCloseToObstacle* testing if

the robot is close to obstacle. If this condition is evaluated true, the transition `t1` is fired and appropriate commands are performed—*stop* and *turnRight* the robot. This, the transition `t1` models a behavior described in point 3.1 in Figure 4.

A short remark to OOPN notation. To record a sequence of transitions (i.e., events that happened in one scenario), we will type `<tName1,tName2,...>`. For instance, `<t1,t2,t3>` means, that the robot came at an obstacle and there is no possibility to go—the robot stops.

VI. System Behavior Modeling

As it was mentioned in Section IV, the modeling process is split up into three basic phases of identification of model elements, modeling the system architecture, and modeling the system behavior. The system behavior modeling will be presented in this section. It will be demonstrated on the previously instroduced case study.

A. Modeling phases cohesion

The important feature is the modeling phases cohesion. Phases are not separated but should be being provided simultaneously. It means, that model elements are finding and improving continuously.

If we analyze the activity net *ExecuteScenario* shown in Figure 6, we can see, that *isCloseToObstacle*, *isClearRoad*, *stop*, *go*, and *turnRight* are commands. Of course, these commands have to have their receiver communicating with a real (or simulated) robot—so, by specification of use case behavior, we have identified a new role in the system—the *Robot*. It also implies that 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. In our example, the role supplies an information about the robot and allows to send commands to it.

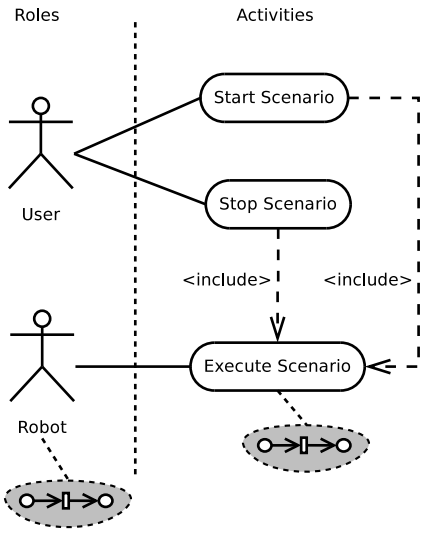

Figure 7. Use Cases of designed system.

So far, we did not take a care about two parts of the scenario—start and stop the robot (see points 1 of basic sequence and point 1 of alternative sequence in Figure 4). Because the robot behavior is an autonomous activity, and use cases we can make a decision to model start and stop as two separated activities. The updated model of use cases are shown in Figure 7. It introduces new role *Robot* and new use cases *Start Scenario* and *Stop Scenario*.

B. Activity Nets

We have presented the activity net *ExecuteScenario* (see Figure 6). The robot can be in two stable states—walking or stopped (there is no possibility to walk). Each such a state is represented by appropriate place, i.e., places `walking` and `stopped`. We have to be able to test activity states, therefore the predicates are generated for each such a place—the synchronous port `isStopped` and the negative predicate `isNotStopped` for the state `stopped` and similar predicates for the state `walking`. Test predicates are shown in Figure 8.

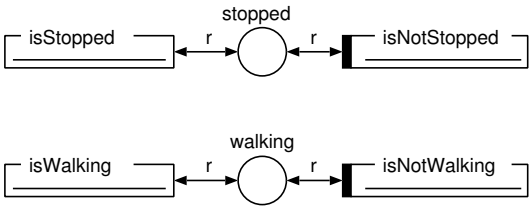

Figure 8. Activity Net *Scenario* – predicates.

Uses cases *Start Scenario* and *Stop Scenario* have to be modeled too. These use cases can be modeled in two ways—in a special activity net or in a method of existing activity net. Because these use cases work only with the activity net *ExecuteScenario*, how deduced from use case diagram in Figure 7, we can model them as methods. So, we have two activities that are using onother activity what is consistent with the layered architecture (Figure 2).

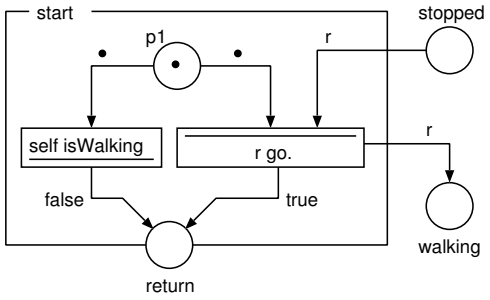

Figure 9. Activity Net *ExecuteScenario* – a method net *start*.

The use case *Start Scenario* is modeled by method net `start` shown in Figure 9. A decision what has to be done is based on the activity state—if the state is *walking* (tested by synchronous port `isWalking`), the method does nothing; if the state is *stopped*, it starts the robot's walk, i.e., sends a message `go` and moves the state token *role* from the place `stopped` to the place `walking`. The state *stopped* is not tested by a predicate, but the transition is directly conditioned by the place `stopped` because it will process the state token `role` in the case of success.

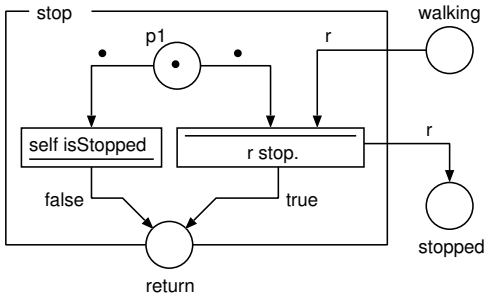

Figure 10. Activity Net *ExecuteScenario* – a method net *stop*.

The use case *Stop Scenario* is modeled by method net `stop` shown in Figure 10. It is similar to the method `start` having following differences. If the state is *stopped*, it does nothing. If the state is *walking*, it sends a message `go` and moves the state token *role* from the place `walking` to the place `stopped`.

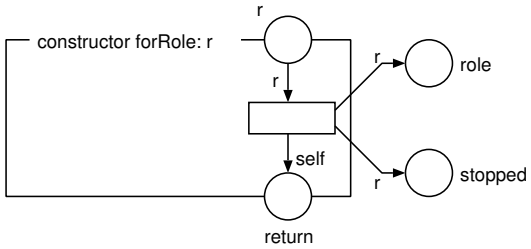

Figure 11. Activity Net *Scenario* – the constructor.

Each activity net is instantiated for just one role, so that the role is initialized by means of constructor as shown in Figure 11. The constructor has one parameter—the role the activity net is assigned to. So each activity has a place storing the role object; the presented constructor has the same structure for every activity nets. In addition, the constructor initializes activity—in our example, it puts a state token *role* to the place *stopped*. After initialization, the robot is in the state `stopped` (the robot stops).

C. Role Nets

A possible model of the role *Robot* is shown in Figure 12. The role checks actual distance of robot to an obstacle each 10 time units (the transition `t1`) 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* (the transition `t2`).

Much like for the activity nets, each role net is initialized by means of the constructor having one parameter—the subject the role net is assigned to. Each activity has a place `subject` storing the subject. The constructor is shown in Figure 13 including the method `turnRight`—it only delegates the message to the subject. Other methods (*go* and *stop*) are modeled in similar way and are not shown.

Each role has its own set of activities it can participate in. The activity nets should be created by asking roles, not

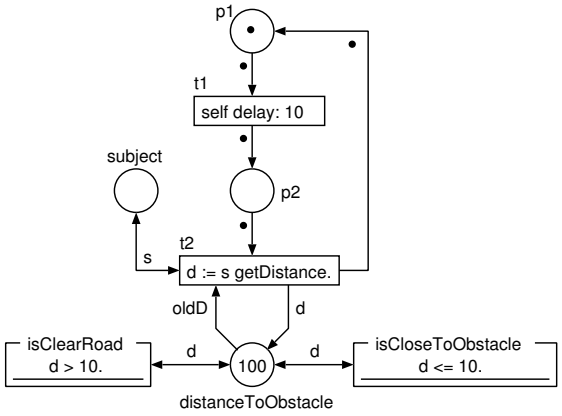

Figure 12. Object net of the role *Robot*.

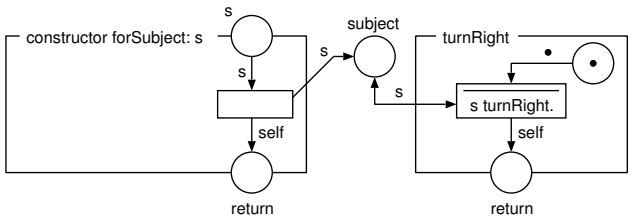

Figure 13. Methods of the role *Robot*.

directly. For instance, the role *Robot* has only one activity net *ExecuteScenario*, so that there is the method *createActivity*, which creates a new instance of activity net *ExecuteScenario*. The method is shown in Figure 14.

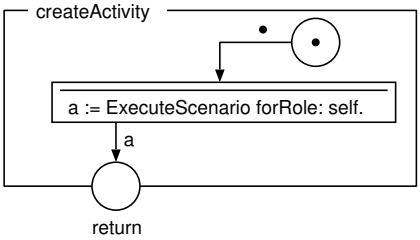

Figure 14. Activity creation of the role *Robot*.

D. Subject Models

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 a database, another system or remote object. The subject can access the real system or can simulate the real system for the testing reason. The subject can be described by the same formalism as activity nets (i.e., by OOPN) or implemented in any language present in the product environment.

It this section, we will demonstrate usign OOPN for subject modeling. The subject for the role *Robot* is named *RobotDevice* and is shown in Figure 15. It represents simulated interface to the real robot in the system. The current distance to

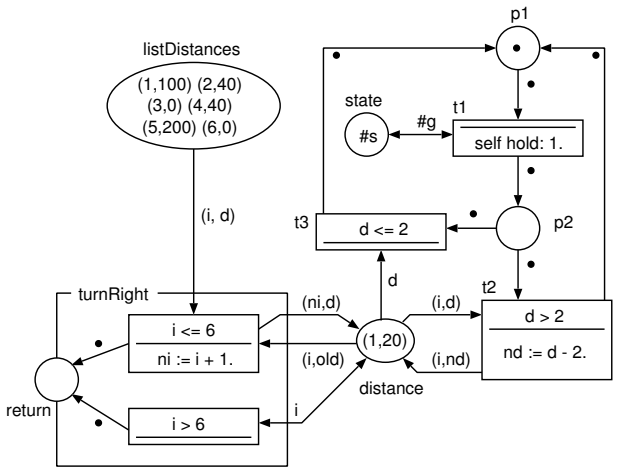

Figure 15. The subject *RobotDevice* – implementation with OOPN.

an obstacle is stored in the place `distance`. If the robot is instructed to go (i.e., the symbol `#g` is stored in the place `state`), the distance is decreased each one time clock (a sequence `<t1,t2>`). If the distance is less than 2 (it simulates the robot is by an obstacle), the distance does not change (a sequence `<t1,t3>`).

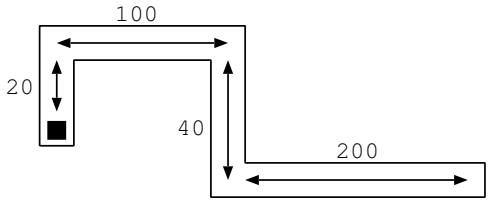

Figure 16. Labyrinth simulated by the subject *RobotDevice*.

The simulated labyrinth the robot is moving in is shown in Figure 16. Simulated robot device contains a list of distances to obstacles (the place *listDistances*), which are subsequently set if the method *turnRight* is performed. First distance is `20`, then it subsequently changes to following values:

- `100` – an item `(1,100)`; the robot turns right, there is a free road 100 units long
- `40` – an item `(2,40)`; the robot turns right, there is a free road 40 units long
- `0` – an item `(3,0)`; the robot turns right, there is an obstacle (a wall)
- `40` – an item `(4,40)`; the robot turns right, there is a road the robot went through (40 units long); in fact, this particular information will not be taken into account, because the previous distance was `0` and the activity net *ExecuteScenario* performs an operation *turn round*, which is implemented by calling *turnRight* twice—the item `(4,40)` will be skipped (see the activity net *ExecuteScenario* in Section V-B)
- `200` – an item `(5,200)`; the robot turns right, there is a free road 200 units long
- `0` – an item `(6,0)`; the robot turns right, there is an obstacle; any other actions invoke no changes

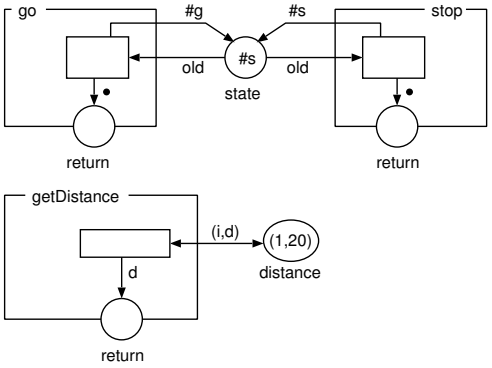

Figure 17. The subject *RobotDevice* – implementation with OOPN.

Methods `go`, `stop`, and `getDistance` is shown in Figure 17. The method `go`, respectively `stop`, puts a symbol `#g`, respectively `#s`, to the place `state`. The method `getDistance` gets a value from the place `distance`.

VII. Architecture Modeling

The way how to model system elements influences the system architecture. Basic architecture is based on pure object oriented approach consisting of classes and relationships between classes. DEVS architecture is based on components that are connected using the formalism of DEVS.

A. Basic Architecture Modeling

Figure 18 shows the classes of basic architecture of our example. Classes from different levels are identified with appropriate stereotypes—*Activity Net*, *Role*, and *Subject*. Each class can be modeled in different formalism, therefore the stereotypes of model formalism are introduced too. In this example, classes are modeled only using Petri nets, so that the stereotype *PN* is used.

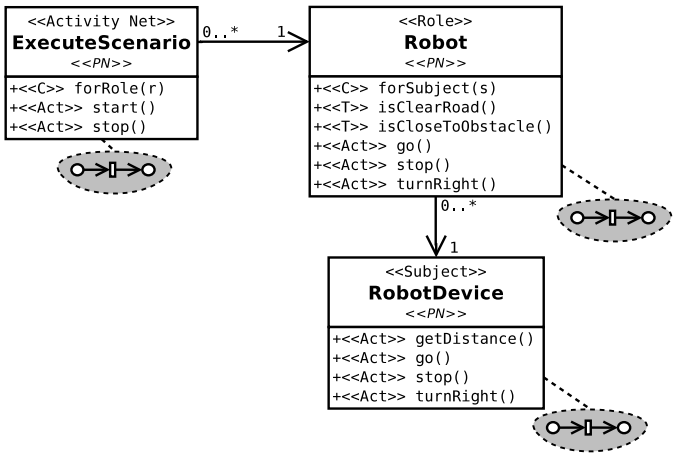

Figure 18. Basic architecture of the case study.

The example 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), `Act` (activity), and `T` (testing) determining a realization of methods (it was introduced in [18]).

B. Combination of Formalisms of DEVS and OOPN

Discrete Event System Specification (DEVS) [19] 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. Let us have the structured set $X = (V_X, X_1 \times \cdots \times X_n)$, where V_X is an ordered set of n variables and $X_1 \times \cdots \times X_n$ denotes a value for each member from the set V_X. We can write the structured set as $X = \{(v_1^x, \ldots, v_n^x) | v_1^x \in X_1, \ldots, v_n^x \in X_n)\}$. Members $v_1^x, \ldots, v_n^x$ are called *input ports*, resp. *output ports* for the set of output events Y. $V_X(v_1^x)$, resp. $V_Y(v_1^y)$, then denotes a value of the input port v_1^x, resp. output port v_1^y. In another words, components are connected by means of ports and event values are carried via these ports.

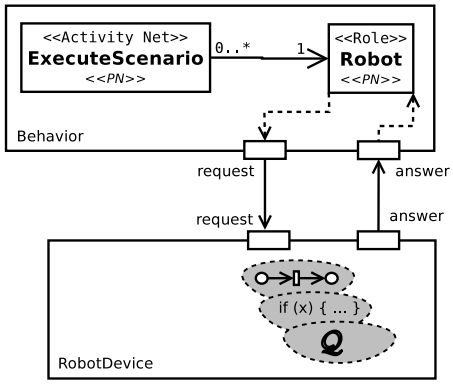

Figure 19. DEVS architecture of the case study – the model *DEVSRobot*.

DEVS components can be described by any formalisms with respecting DEVS functions. Thus, 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.

C. DEVS Architecture Modeling

The model can be split up into components in accordance to their responsibility. For instance, the system of our case study has two basic parts (i.e., components)—the model of behavior (activity nets and roles) and the model of real robot (subjects; the subject can provide a communication channel to a real robot or can simulate it). Components can be modeled using means of UML, i.e., packages. In that case, the component interface is provided by classes themselves, so that the replacement of components is complicated. Formalisms such as DEVS define a stable interface allowing to exchange components in a very simple way, because components are connected only by means of ports. It is one of reasons we have made a decision to use DEVS formalism to describe the system architecture.

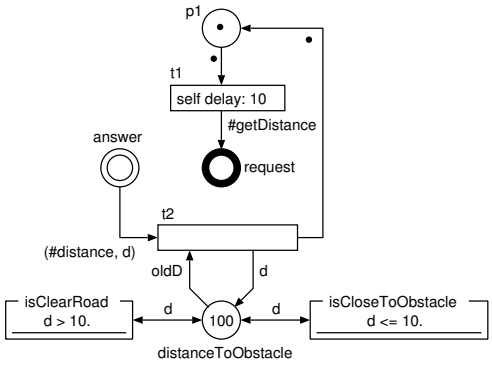

Figure 20. The role *Robot* – implementation for DEVS architecture.

The DEVS architecture of presented case study contains two components *Behavior* and *RobotDevice* as shown in Figure 19. The component *Behavior* describes the system behavior, as presented in a case of basic architecture. The component *RobotDevice* describes the robot subject and can be modeled by OOPN, programming language, or any other supported formalism. Components are connected via ports *request* and *answer*. Both components are coupled the model called *DEVSRobot*.

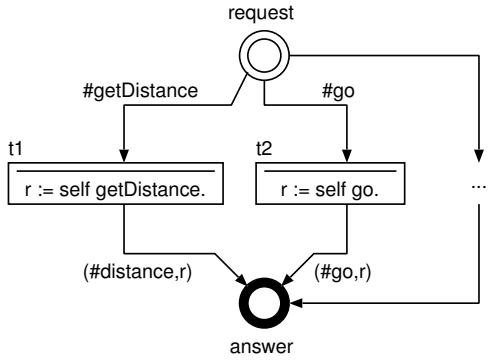

Figure 21. Extension of the subject model *RobotDevice* for the DEVS architecture reason.

Because the architecture changes, we have to modify classes describing system behavior in the component *Behavior*. This component encapsulates OOPN model and the initial class has to be defined. Because the interface between components serves for communication to the subject of robot, and subjects can communicate to roles, the *Robot* class will be the initial one. It means, that ports *request* and *answer* are mapped to places of the *Robot* object net. This modified object net is shown in Figure 20. Place named *request*, resp. *answer*, corresponds to output port *request*, resp. input port *answer*.

$$
\begin{aligned}
variables: \quad & r \leftarrow nil \\
& d \leftarrow 20 \\
& dists[] \leftarrow array\ of\ (100, 40, 0, 40, 200, 0) \\
& i \leftarrow 0 \\
& go \leftarrow false \\
getDist(): \quad & return\ (\#distance, d) \\
turnR(): \quad & i < 6 : i \leftarrow i + 1 \\
& d \leftarrow dists[i] \\
go(): \quad & go \leftarrow true \\
stop(): \quad & go \leftarrow false
\end{aligned}
$$

Figure 22. Internal data and functions of the component *RobotDevice*.

The component *RobotDevice* can be described by OOPN. The posible model results from the class *RobotDevice* shown in Figures 15 and 17. This class is an initial class of the component and adds model elements to the object net as shown in Figure 21. It gets a request string from its input port $request$, asks itself for answer, and puts the answer to its output port $answer$.

Another way is to describe the component *RobotDevices* as an atomic DEVS. First, the internal data and functions are defined as shown in Figure 22. They correspond to the methods and data stored in places of basic model of the subject *RobotDevice* shown in Figures 15 and 17.

DEVS functions are defined as shown in Figure 23. They use internal data and functions. If an external event occurs, the function δ_{ext} is performed; it puts a value from input port *request* to the variable r. Time advance function ta is defined as follows: if there is a request ($r \neq nil$), then internal and output functions will be called immediatelly ($ta \leftarrow 0$); if there is no request and the variable go is $true$, then internal and output functions will be called in 1 time unit ($ta \leftarrow 1$); if there is no request and the variable go is $false$, then nothing happen ($ta \leftarrow \infty$). Output function λ calls internal functions depending on the request r. If called functions return any value a, this value a is put to the output port *answer*. If the internal transition function δ_{int} is called and there is no request, it decrease the distance to an obstacle ($d \leftarrow d - 2$). In any case, it destroys an information about processed request ($r \leftarrow nil$).

VIII. Software Deployment with Models

This section will demonstrate possibilities of keeping models in the deployed system. The goal is to use the same formalism for system modeling as well as for system implementation and deployment. It is based on the application framework allowing to interoperability of models and product environment.

$$
\begin{aligned}
\delta_{ext}: \quad & r \leftarrow V_X(request) \\
\lambda: \quad & r = \#getDistance : a \leftarrow getDist() \\
& r = \#turnRight : a \leftarrow turnR() \\
& r = \#go : a \leftarrow go() \\
& r = \#stop : a \leftarrow stop() \\
& a \neq nil : V_Y(answer) \leftarrow a \\
\delta_{int}: \quad & r = nil\ \&\ d > 2 : d \leftarrow d - 2 \\
& r \leftarrow nil \\
ta: \quad & \begin{cases} 0, & r \neq nil \\ 1, & go\ \&\ r = nil \\ \infty, & not\ go\ \&\ r = nil \end{cases}
\end{aligned}
$$

Figure 23. DEVS functions of the component *RobotDevice*.

A. Application Framework

The application framework has to fulfil two basic requirements. First, to link models and product environment. Second, to work with models in simulations.

First, the models described by means of OOPN can cooperate with objects of the product environment (product objects). Since the developed framework [20] is implemented in Smalltalk [21], 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).

Second, 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.

B. Implementation with Basic Architecture

We can exchange the simulated subject by an interface to the real robot. It is very simple—we only create instances of appropriate classes and do not care about used formalism. Figure 24 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:
                    'ExecuteScenario'.
cRole := Repos componentNamed: 'Robot'.
subj := RobotDevice new.
role := cRole forSubject: subjR.
actS := role createActivity.
actS start.
```

Figure 24. Accessing OOPN objects from Smalltalk.

Now, we demonstrate an accessing OOPN objects from product environment of Smalltalk. We send a command `go` to start walking—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. We obtain a special meta-protocol by sending a message `asPredicate` and then call synchronous port or negative predicate in the standard form of message passing. 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 25.

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

Figure 25. 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 26. 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 26. Waiting for a condition.

Second way is shown in Figure 27. 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 27. 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 28.

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

Figure 28. Listener implementation.

C. Implementation with DEVS Architecture

The example of accessing DEVS components and their object interface is shown in Figure 29. First, we get the DEVS model named *DEVSRobot*, which is based on architecture from Figure 19. Second, we obtain DEVS component *Behavior*, which is able to communicate through its ports. Since this component is described by OOPN, 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: 'DEVSRobot'.
cB := c1 componentNamed: 'Behavior'.
role := cB objectInterface.
```

Figure 29. Obtaining object interface to the inital object.

The variable `role` referees an instance of initial class *Robot* of the component *Behavior*. The other manipulation is the same as in the case of the basic architecture. For instance, to wait for the condition *isCloseToObstacle* a sequence of messages shown in Figure 26 can be used. It blocks until the port `isCloseToObstacle` becomes fireable and then stops the robot.

IX. Conclusion and Future Work

The paper dealt with an approach to system development and deployment using formal models as a basic design, analysis and programming means combining simulated and real components. Combination of two formalisms has been taken into account—Object Oriented Petri Nets (OOPN) for behavior description and Discrete Event System Specification (DEVS), which can be used for architecture description as well as behavior description. The main advantage of that approach is no need for code generation and further investigation of deployed systems using the same formal models. The process of such an development was demonstrated on the case study of simple robotic system.

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] R. Kočí and V. Janoušek, "Object oriented Petri nets in software development and deployment," in ICSEA 2013, The Eighth International Conference on Software Engineering Advances. Xpert Publishing Services, 2013, pp. 485–490.

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

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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, 2008, pp. 55–60.

[13] 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.

[14] D. Moldt, "OOA and Petri nets for system specification," in Object-Oriented Programming and Models of Concurrency. Italy, 1995.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] R. Kočí, "PNtalk system," http://perchta.fit.vutbr.cz/pntalk2k, 2004. [Online]. Available: http://perchta.fit.vutbr.cz/pntalk2k

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