Validation of Specification Models Based on Petri Nets

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*—Each validation process of the software system requirements should include an analysis of all possible scenarios. Whereas only some of them are valid, some scenarios are redundant, and some scenarios cause unsafe behavior of the system. An important factor for successful checking of all possible scenarios is the appropriate support of searching and evaluation of scenarios. In this area, there is a gap between what formal approaches can offer and how they are actually used. It comes from the belief that formal approaches are difficult for understanding and using, and that they are not suitable for validation because they have no executable form. Nevertheless, systematic formal description techniques allow to specify the system properties and the detailed form of the solution during the design process and to analyze system specification, including user interactions, and implement architectural design decisions. This work focuses on the use of Petri nets for specifying requirements and generating and analysis scenarios to validate this specification.**

Keywords–Object Oriented Petri Nets; Use Cases; Sequence Diagrams; requirements specification; requirements validation.

I. Introduction

This work builds on the paper [1] and describes possible validation procedures for the specification models. It is part of the *Simulation Driven Development* (SDD) approach [2], which combines basic models of the most used modeling language Unified Modeling Language (UML) [3][4] and the formalism of Object-Oriented Petri Nets (OOPN) [5].

One of the fundamental problems associated with software development is the specification and validation of the system requirements [6]. The use case diagram from UML is often used for requirements specification, which is then developed by other UML diagrams [7]. The disadvantage of such an approach is an inability to validate the specification models and it is usually necessary to develop a prototype, which is no longer used after fulfilling its purpose. Utilization of OOPN formalism enables the simulation (i.e., to execute models), which allows to generate and analyze scenarios from specification models. All changes enforced during the validation process are entered directly in the specification model, which means that it is not necessary to implement or transform models.

There are methods of working with modified UML models that can be transformed to the executable form automatically. Some examples are the MDA methodology [8], Executable UML (xUML) [4] language, or Foundational Subset for xUML [9]. These approaches are faced with a problem of model transformations. It is hard to transfer back to model all changes that result from validation process and the model becomes useless. Further similar work based on ideas of model-driven development deals with gaps between different development stages and focuses on the usage of conceptual models during the simulation model development process [10]. This approach is called *model continuity*. While it works with simulation models during design stages, the approach proposed in this paper focuses on *live models* that can be used in the deployed system.

The paper is organized as follows. Section II summarizes concepts of the design method with using use cases and Petri nets. It also introduces the simple case study. Section III demonstrates possibilities of recording scenarios based on Petri nets. Section IV deals with scenarios exploration including generating scenarios and sequence diagrams. The summary and future work is described in Section V.

II. Design Method

In this section we will briefly introduce basic concepts of the design method [11] and will demonstrate these concepts on a simple case study.

A. *Case Study in Basic Diagrams*

The basis of design method is to identify use cases and roles that interact with individual use cases; the use case diagrams from the UML language are used. The behavior of use cases and roles are described by special variant of Petri nets, Object-Oriented Petri Nets (OOPN). Use cases and roles correspond to classes of OOPN. One use case invocation corresponds to creating an instance, i.e., an object of the class. The basic behavior of each element is described by one object net that represents independent autonomous behavior of the object. Because the life of object and its object net is closely related, we can use the notion *object* and *net* in the same meaning. The object net, i.e., the basic behavior, can be supplemented with method nets. Nets describing behavior of roles are called *role nets*, nets describing behavior of use cases are called *activity nets*.

The case study consists of simplified robotic system. For our purposes, we will consider only one robot whose motion is controlled by a predefined algorithm. So that the model consists of one role of *Robot* and one use case *Algorithm1*. The actor *Robot* represents a role of the real robot in the system.

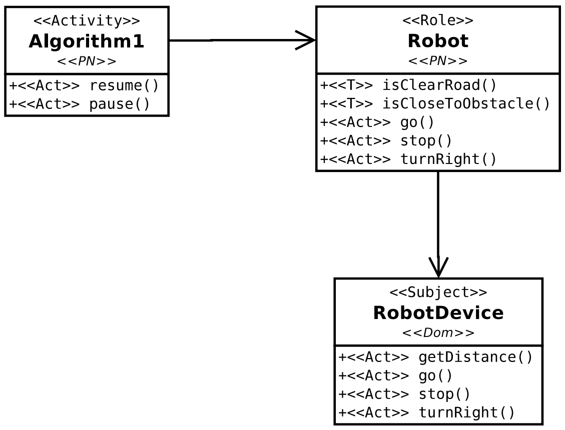

Figure 1. The basic class diagram.

The real actor, i.e., robot, has to have own representation is the system too. For terminological reasons we denote a real actor by the term *subject*. The actor *Robot* has its subject called *RobotDevice*. The classes of role, activity, and subject nets are shown in Figure 1. A more detailed description of the model can be found in the paper [1].

B. Behavioral modeling

Each object net, i.e., use case or role specification, describes a set of scenarios of the same type. From the Petri nets definition, the common behavior is defined as an oriented graph consisting of two kinds of nodes, transitions and places. Transitions representing actions or commands and places representing partial states of the scenario. Only nodes from different kinds can be connected by arcs.

The system state is represented by places of the nets. System is in a particular state if an appropriate place contains a *token*. Actions that can be performed in a particular state are modeled as part of the transition whose execution is conditioned by a presence of tokens in that state. The transition is modeled as an element that moves tokens between places, i.e., particular states. Except the input places, the transition firing can be conditioned by a *guard*. The guard contains expressions resulting in boolean value. The expression may also be a synchronous port call. Synchronous port is a special variant of transition, i.e., it may have input places, output places, and a guard. Synchronous port cannot be fired independently but has to be called from another transition or sycnhronous port. These ports serve for synchronous communication between nets, i.e., calling transition and called port have to be fired simultaneously.

The transition can be fired only if the guard is evaluated as true. It means that every boolean expression gets true and every called synchronous port gets fireable. If the transition fires, it executes all called synchronous ports that can have a side effect, i.e., the executed synchronous port can change a state of the called net.

C. Activity Net Algorithm1

The activity net *Algorithm1* of the use case *Algorithm1* (see Figure 2) consists of states *testing*, *walking*, *stopped*, *turnRight*, and *turnRound* that are represented by appropriate places. States *turnRight* and *turnRound* are only temporal and the activity goes through these states to the one of stable states *walking* or *stopped*. This net describes the following algorithm. The robot goes straight and if it encounters an obstacle, it turns to the right and tries to go straight. If it can not go straight, it turns around. If it can not go straight, it stops.

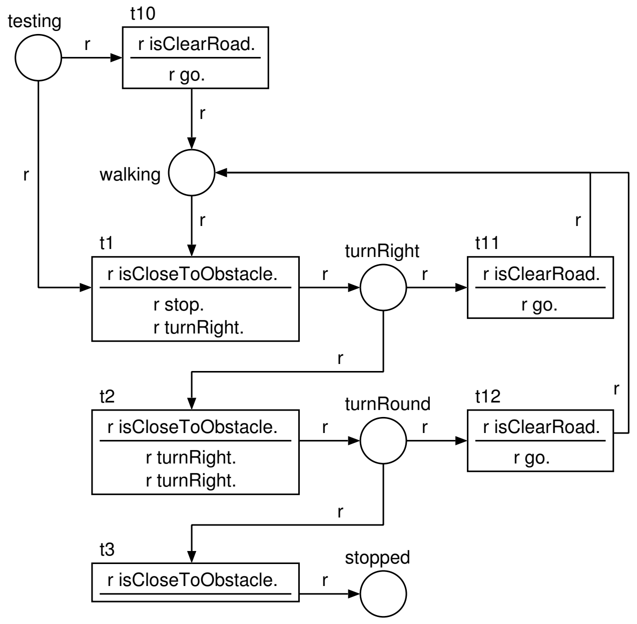

Figure 2. Model of the use case *Algoritm1*.

Control flow is modeled as the sequence of transitions. Each transition execution is conditioned by events representing the state of the robot. Let us take one example for all, the state *testing* and linked transitions *t10* and *t1*. The transition *t1* is fireable, if the condition *isCloseToObstacle* is met. This condition is modeled by calling the synchronous port in the guard. When firing the transition, actions *stop* and *turnRight* the robot are performed and the system moves to the state *turnRight*. The transition *t10* is fireable, if the condition (modeled by the synchronous port) *isClearRoad* is met. When firing the transition, the action *go* (the robot goes straight) is performed and the system moves into the state *walking*.

Both testing condition and message passing represent the interaction between the system (especially the activity net *Algorithm1*) and the role of robot (the role net *Robot*). The object of the robot role serves as token moving through the control flow. Presence of this token in places represents particular states and allows the activity net to communicate with the robot at the same time.

D. Role Net Robot

As already mentioned, actor represents a *role* of the user or device (i.e., a real actor), which the actor can hold in the system. One real actor may hold multiple roles, so that it can be modeled by various actors.

A role is modeled as a use case and its behavior by Petri nets. Interactions between use cases and actors are synchronized through *synchronous ports* that test conditions, convey

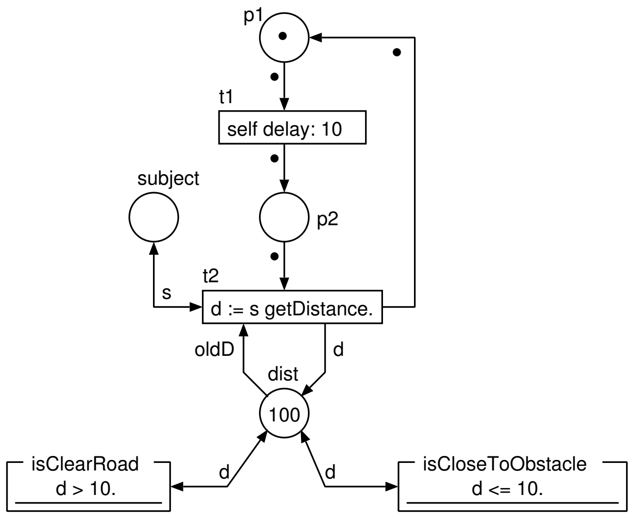

Figure 3. Model of the role net *Robot*.

the necessary data and can initiate an alternative scenario on both sides. For instance, the robot state is tested by a pair of synchronous ports *isClearRoad* and *isCloseToObstacle*.

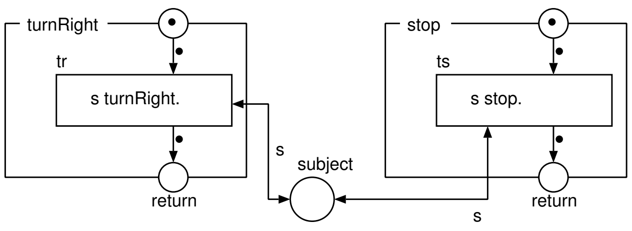

Figure 4. Methods of the role net *Robot*.

The net can send or receive instructions through messages too. In our example, the role *Robot* checks the distance from an obstacle each 10 time unit by sending the message *getDistance* to the robot subject. Methods *turnRight*, *go*, and *stop* that control the robot moving are delegated to the subject. They are shown in Figure 4.

III. Modeling of Scenarios

Petri nets models describe possible scenarios of one type of behavior, i.e., a behavior of a use case or an actor. For testing purposes it is necessary to investigate specific scenarios for individual situations. This section demonstrates possibilities of recording individual scenarios based on Petri nets models.

A. Scenario records

To record one scenario, we use the notation common to the Petri nets, the sequence of fired transitions. The basic notation of the record is $<\mathrm{tName1}, \mathrm{tName2}, \ldots>$. For instance, the record $<\mathrm{t10}, \mathrm{t1}, \mathrm{t11}>$ means that the robot will go straight, after a while it encounters an obstacle, turns right, and continuous walking. The record may be complemented with data including place markings and constraints. The previous example may be complemented with place markings after the scenario ends. At this moment, all places are empty except the place $\mathrm{walking}$, which contains object of the class Robot as the control token. The sequence is $<\mathrm{t10}, \mathrm{t1}, \mathrm{t11}, \ldots, \mathrm{t3}\{\mathrm{stopped}(\trianglelefteq\mathrm{Robot})\}>$, where the notation $\trianglelefteq\mathrm{Robot}$ means *an instance of the class Robot*. If it is needed to name the instance, we will write $\mathrm{name}\trianglelefteq\mathrm{Robot}$.

B. Subject Model

To have the model complete, we will simulate the subject representing the real robot and the environment the robot is moving in. We come out of the labyrinth model, which is shown in Figure 5. The subject *RobotDevice* is modeled by the Petri net, which is captured in Figure 6.

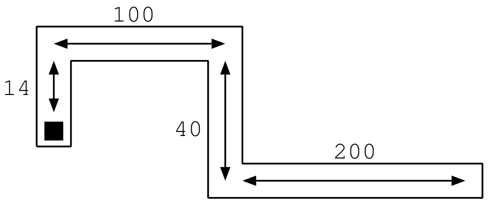

Figure 5. Labyrinth scheme.

The model has two places representing stable states of the net—state (the symbol in this place indicates whether the robot walks or not) and $\mathrm{distance}$ (the pair of numbers represents the position in labyrinth and the distance from an obstacle). If the net is in the state *to go* (symbol $\#\mathrm{g}$ is placed in the state state), the distance from the obstacle is reduced by two length units each time unit. If the robot reaches the obstacle, the distance does not change anymore. The shape of labyrinth is modeled by pairs of values in the place $\mathrm{listDistances}$. The first value denotes the position, i.e., the corridor the robot should walk in, and the second value denotes the length of this corridor. The position changes by calling method $\mathrm{turnRight}$.

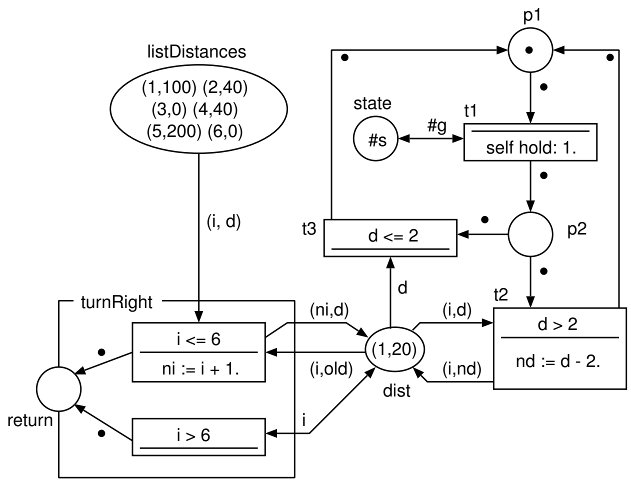

Figure 6. Model of the subject *RobotDevice* simulating the real device.

The formalism of OOPN allows working with time using a special method *delay* that is called from transitions. When transition containing a *delay* message is invoked, this transition is delayed for the specified time. It has the same meaning as the timed transition in Timed Petri Nets. The simulator can interpret the time in two ways. Either it works with model time that simulates real time during simulation run or directly real time.

C. Sequence of events

The transition sequence of the net *Algorithm1* representing the particular scenario, which corresponds to the labyrinth model, is shown in Figure 7. Such listings are well machine-readable, but are less readable for humans. It is possible to graphically record the sequence of the performed transitions, which can include additional information about the selected states or time.

$$<\mathrm{t10}, \mathrm{t1}, \mathrm{t11}, \mathrm{t1}, \mathrm{t11}, \mathrm{t1}, \mathrm{t2}, \mathrm{t12}, \mathrm{t1}, \mathrm{t2}, \mathrm{t3}>$$

Figure 7. The transition sequence of the *Algorithm1* net.

An example of the graphical notation of the record is shown in Figure 8. The record is a sequence of fired transition with no conditions and branches. The information about chosen places are displayed above arcs before and after the transition fires. For our purposes, we have chosen the place $\mathrm{Robot.dist}$ (first line, e.g., (100)) and the place $\mathrm{RobotDevice.dist}$ (second line, e.g., $(1, 20)$). Each record of fired transitions can be supplemented by an information about model time (e.g., $\mathrm{t} = 0$).

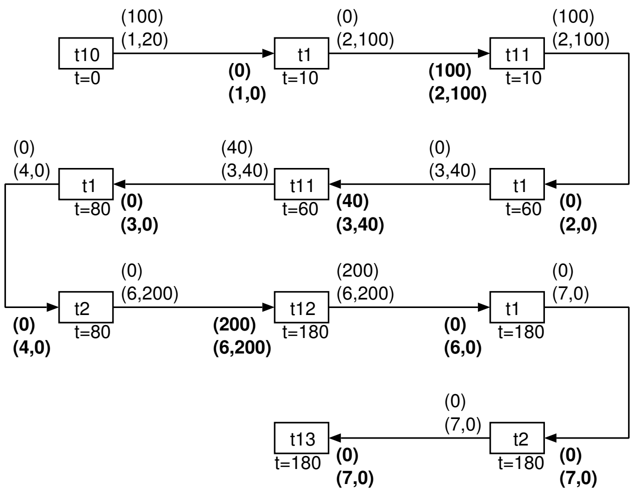

Figure 8. Graphic record of the expected scenario.

The transition sequence can be recorded manually or automatically. The first approach serves as a test case, which is compared to the sequence obtained by model simulation. In the case of manual recording, it is not advisable to declare states and model time for all transitions, but only for significant points in the sequence of transitions. In our example, there are important locations before performing transitions $\mathrm{t11}$, $\mathrm{t12}$, and $\mathrm{t13}$. Figure 9 shows the initial part of the declared sequence (show at the top) and the obtained (real) sequence (shown at the bottom). By comparison, we can find out that the real sequence differs from expected sequence in the third step (transition).

IV. Exploration of Sequences

In this section, we will explore the difference between expected and obtained sequences. Since the model simulation is not limited to one net, we have to take into account the behavior of other interconnected networks. Therefore, we will analyze transitions over time across all participating nets.

A. Records of sequence

To save space, we will not display sequences of events graphically, but describe them in a table. The table record includes model time t, fired transitions trans, states of chosen places $\mathrm{Alg1.walking}$ (p1), $\mathrm{Alg1.turnRight}$ (p2), $\mathrm{Alg1.turnRound}$ (p3), $\mathrm{Robot.dist}$ (Rdist), $\mathrm{RobotDevice.dist}$ (Ddist), and $\mathrm{RobotDevice.state}$ (Dstate).

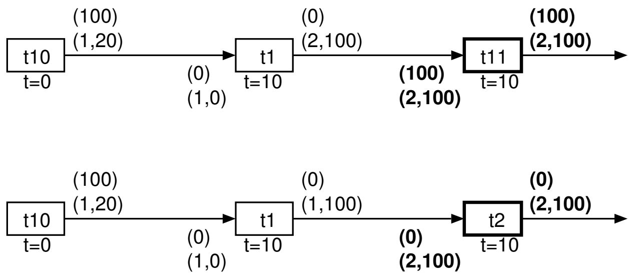

Figure 9. Graphic record of the declared and obtained scenarios.

TABLE I. SEQUENCE OF EVENTS OF THE BASIC SCENARIO.

t	trans	p_1	p_2	p_3	Rdist	Ddist	Dstate
0	Alg1.t10	r			100	$(1, 20)$	#g
	Robot.t1	r			100	$(1, 20)$	#g
	RDev.t1	r			100	$(1, 20)$	#g
1	RDev.t2	r			100	$(1, 18)$	#g
	RDev.t1	r			100	$(1, 18)$	#g
2	RDev.t2	r			100	$(1, 16)$	#g
…	…	…	…	…	…	…	…
9	RDev.t2	r			100	$(1, 2)$	#g
	RDev.t1	r			100	$(1, 2)$	#g
10	RDev.t2	r			100	$(1, 0)$	#g
	Robot.t2	r			0	$(1, 0)$	#g
	<S>Alg.t1				0	$(1, 0)$	#g
	Robot...t				0	$(1, 0)$	#g
	RDev...t				0	$(1, 0)$	#s
	Robot...t				0	$(1, 0)$	#s
	RDev...t				0	$(2, 100)$	#s
	<F>Alg.t1		r		0	$(2, 100)$	#s
	<S>Alg.t2				0	$(2, 100)$	#s
	Robot...t				0	$(2, 100)$	#s
	RDev...t				0	$(3, 40)$	#s
	Robot...t				0	$(3, 40)$	#s
	RDev...t				0	$(4, 0)$	#s
	<F>Alg.t2			r	0	$(4, 0)$	#s
	Alg.t3				0	$(4, 0)$	#s

Table I shows the sequence of transitions from the beginning of the simulation, i.e., from the $\mathrm{Alg1.t10}$ transition. We find out that the transitions of nets Robot and RDev are performed between the transitions $\mathrm{Alg1.t10}$ and $\mathrm{Alg1.t1}$ of the base sequence. Sequence of these transitions simulates the robot movement and updates the distance information. Transition $\mathrm{Alg.t1}$ is activated when information of the distance is updated to value of 0. This activation corresponds to the line with <S>Alg.t1 symbol. When this transition fires, the transitions of Robot.stop and RDev.turnRight nets are performed. After completing the Alg1.t1 transition, the system is in a state that is captured on the line with <F>Alg.t1 symbol. The next step is an activation of the Alg1.t2 transition even though the declared sequence of events expects an activation of Alg1.t11. There is a problem because the Alg.t1 transition did not change the condition guarding transitions Alg.t2 and Alg.t11.

TABLE II. SEQUENCE OF EVENTS OF THE CORRECTED SCENARIO.

t	trans	p_1	p_2	p_3	Rdist	Ddist	Dstate
0	Alg1.t10	r			100	(1, 20)	#g
	Robot.t1	r			100	(1, 20)	#g
	RDev.t1	r			100	(1, 20)	#g
1	RDev.t2	r			100	(1, 18)	#g
	RDev.t1	r			100	(1, 18)	#g
	Robot.t2	r			18	(1, 18)	#g
	Robot.t1	r			18	(1, 18)	#g
2	RDev.t2	r			18	(1, 16)	#g
...	...	...	...	...	...	...	...
5	RDev.t2	r			12	(1, 10)	#g
	RDev.t1	r			12	(1, 10)	#g
	Robot.t2	r			10	(1, 10)	#g
	Robot.t1	r			10	(1, 10)	#g
	<S>Alg.t1				10	(1, 10)	#g
	Robot...t				10	(1, 10)	#g
	RDev...t				10	(1, 10)	#s
	Robot...t				10	(1, 10)	#s
	RDev...t				10	(2, 100)	#s
	<F>Alg.t1		r		10	(2, 100)	#s
	<S>Alg.t2				10	(2, 100)	#s
...	...	...	...	...	...	...	...

According to the state analysis, it can be deduced that the information in the *Robot* net about distance to the obstacle has not been updated. In addition, there is a long delay in passing the current information from the subject *RobotDevice* to the role *Robot*. If we focus on this problem, the solution is relatively simple. We need to change the interval in which the information is obtained so that the response is faster. We change the action of Robot.t1 transition to the self hold: 1 statement. The resulting sequence is shown in Table II. The information is being updated but the previous issue is not addressed—the actual scenario is still different from the expected one. We will analyze this situation in next subsections.

B. Sequence Diagrams

It is not easy to get an overview of the communication between objects in large models. One scenario corresponds to a sequence of interactions between system objects. Interactions are usually described by diagrams. The activity diagram and the sequence diagram of the UML language being widely used in this area. The activity diagram is suitable for modeling the behavior of the use case, i.e., modeling all possible scenarios in general way. The sequence diagram models one particular scenario and makes it possible to better represent the external view of the system dynamic, i.e., the messaging sequence. We will present the possibilities of using sequence diagrams in conjunction with Petri nets.

The Petri net model is conceived as a sequence of internal and external events. Internal events may represent message sending to another objects, external events may arise in response to incoming events. Having a classical concept into account, it is necessary to map the external events to methods. Nevertheless, it makes the model less readable and understandable. When using Petri nets, the scenario is clearly defined as a sequence of events. One can then monitor system dynamics directly in the base model without event mapping. On the other hand, the sequence diagram makes it possible to better represent the external view of the system dynamic, i.e., the sequence of messages.

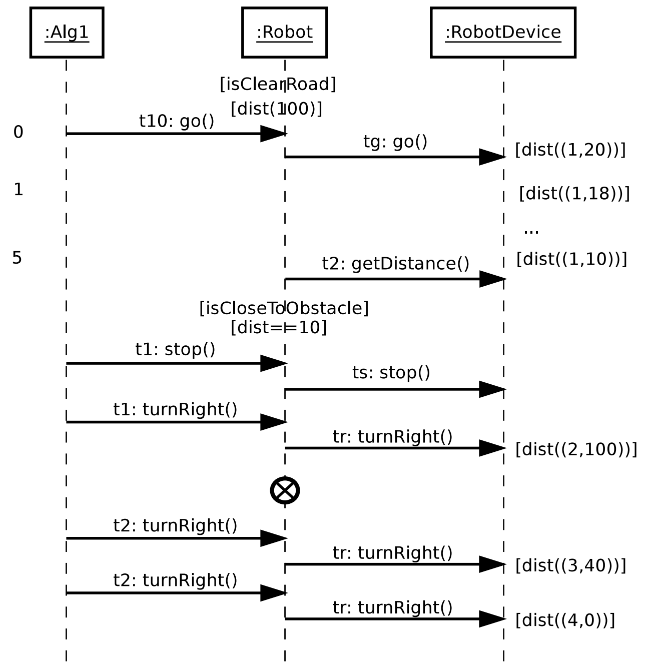

Figure 10. Sequence diagram modeling the scenario.

In addition to statistical data, the information needed to generate the sequence diagram can be collected during the simulation. It is therefore possible to generate individual scenarios in the form of sequence diagrams. The message linked to the event is generated to the sequence diagram as the message between sender and receiver. The synchronous port connected to an event is captured in the sequence diagram as the state of the object on which the port has been executed.

$$<\mathrm{Alg1.t10}, \mathrm{Robot.isClearRoad}, \mathrm{Robot.go.t1}, \mathrm{RobotDevice.go.t1}>$$

Figure 11. Part of the complete transition sequence.

Let us get back to our example. Since we know that the problem occurs before executing the transition Algorithm1.t2, it is sufficient to generate a sequence diagram from the first event Algorithm1.t10 to the event Algorithm1.t2. The resulting sequence diagram is shown in Figure 10. For instance, at time 0, the object $o_1 \trianglelefteq \mathrm{Alg1}$ sends a message go to the object $o_2 \trianglelefteq \mathrm{Robot}$ from the transition Alg1.t10. This execution is conditioned by synchronous port isClearRoad and the initial marking of the place Robot.dist is 100. The object o_2 responds by forwarding the message to $o_3 \trianglelefteq \mathrm{RobotDevice}$ object. This sequence corresponds to the initial part of the scenario shown in Table II, the formal notation is captured in

Figure 11.

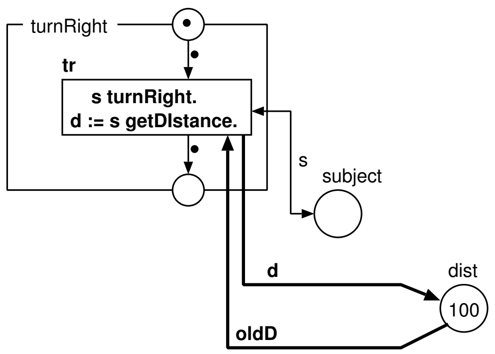

Figure 12. Fixed method of the role net *Robot*.

In Figure 10, there is a special symbol ⊗ marking the position in the sequence where is a difference between expected and obtained scenario. We knew there is a problem of transmitting information about robot's distance. In the sequence diagram, we can find out that the message of getting distance is not called after turning the robot. We fix this by calling the message getDistance inside the transition Robot.tr as shown in Figure 12.

C. Interface to real robot

In the next step, we will analyze the behavior of model connected to the real robot. From the model point of view, only the subject stored in the place Robot.subject changes. Because we work with a real component, we run the simulation in real time rather than model time. The first steps of the simulation are shown in Table III.

TABLE III. SEQUENCE OF EVENTS OF THE OBTAINED SCENARIO.

t	trans	p_1	p_2	p_3	Rdist
0.00	Alg1.t10	r			100
	Robot.t1	r			100
1.03	Robot.t2	r			18
...	...	...	...	...	...
5.10	Robot.t2	r			9
5.10	Robot.t1	r			9
5.10	<S>Alg.t1				9
5.11	Robot...ts				9
5.11	Robot...tr				0
5.11	<F>Alg.t1		r		0
5.12	<S>Alg.t2				0
...	...	...	...	...	...

We have got the same situation—the robot stops prematurely. Looking at the sequence of events, we find out that the robot turns too early and stands in front of the wall of corridor that it came. It is necessary to adjust the role behavior so that it can slow down and gradually stop just before the obstacle.

V. CONCLUSION

The paper dealt with the concept of modeling software system requirements using the formalism of OOPN. This concept allows to model and validate specifications through the scenarios exploration in simulated or real surroundings with no need to transform models. We presented basic concepts based on declaration, generation, and comparison of individual scenarios. The concept is supported by mapping Petri net model to sequence diagrams helping display the sequence of messages. During the process of model analysis, we discovered several inaccuracies in the description of role behaviors, but own algorithm, i.e., the basic work-flow of the system, was not modified. This approach of creating the requirements specification combines an abstract view of the system with implementation details, all of which are implemented by the same formalisms.

At present, we have developed the tool supporting requirements modeling using use cases and the formalism of OOPN. In the future, we will focus on the tool completion, a possibility to interconnect model to others languages, and feasibility study for different kinds of usage.

ACKNOWLEDGMENT

This work has been supported by the internal BUT project FIT-S-17-4014 and The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602.

REFERENCES

[1] R. Kočí and V. Janoušek, "Modeling System Requirements Using Use Cases and Petri Nets," in ThinkMind ICSEA 2016, The Eleventh International Conference on Software Engineering Advances. Xpert Publishing Services, 2016, pp. 160–165.

[2] 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.

[3] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

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

[5] M. Češka, V. Janoušek, and T. Vojnar, "Modelling, Prototyping, and Verifying Concurrent and Distributed Applications Using Object-Oriented Petri Nets," Kybernetes: The International Journal of Systems and Cybernetics, vol. 2002, no. 9, 2002.

[6] K. Wiegers and J. Beatty, Software Requirements. Microsoft Press, 2014.

[7] N. Daoust, Requirements Modeling for Bussiness Analysts. Technics Publications, LLC, 2012.

[8] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in International Conference on Software Engineering, ICSE, 2010.

[9] S. Mijatov, P. Langer, T. Mayerhofer, and G. Kappel, "A framework for testing uml activities based on fuml," in Proc. of 10th Int. Workshop on Model Driven Engineering, Verification, and Validation, vol. 1069, 2013.

[10] D. Cetinkaya, A. V. Dai, and M. D. Seck, "Model continuity in discrete event simulation: A framework for model-driven development of simulation models," ACM Transactions on Modeling and Computer Simulation, vol. 25, no. 3, 2015.

[11] R. Kočí and V. Janoušek, "Formal Models in Software Development and Deployment: A Case Study," International Journal on Advances in Software, vol. 7, no. 1, 2014, pp. 266–276.