Specification of Requirements Using Unified Modeling Language and Petri Nets

Radek Kočí and Vladimír Janoušek

Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence
Czech Republic
email: {koci,janousek}@fit.vutbr.cz

***Abstract*—One of the major problems the software engineering is dealing with is the correct specification and implementation of requirements to the system being developed. A lot of design methods use models of the Unified Modeling Language for requirements specification and further design of the system. To validate the specification, the executable form of models has to be obtained or the prototype has to be developed. This may cause errors in the transformation or implementation process, which results in incorrect validation. The approach presented in this work focuses on formal requirement modeling combining the classic models for requirements specification (use case diagrams and class diagrams) with models having a formal basis (Petri Nets). Created models can be used in all development stages including requirements specification, verification, and implementation. All design and validation steps are carries on the same models, which avoids mistakes caused by model implementation.**

Keywords–Object Oriented Petri Nets; Use Cases; requirement specification; requirement implementation.

I. Introduction

This work is based on the paper [1], which is extended of detailed explanation of modeling requirements and behavior of software systems using formal models. This work is part of the *Simulation Driven Development* (SDD) approach [2] and 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].

The fundamental problem associated with software development is an identification, specification and subsequent implementation of the system requirements [6]. Many design methods have no formal definitions and depend on intuitive approach to requirements specification and design. It results in troubles with correct specification of system requirements and their realization. The second problem is a rather complicated way to verify designed concepts through models under realistic conditions. Designers either have to implement a prototype or transform the models into executable form, which can be tested. All changes that result from testing are difficult to transfer back to models that become useless.

Formal techniques allow to specify system requirements and the solution in a clear, unambiguous way. Nevertheless, there is a gap between the features that formal approaches may offer and how they are actually utilized in the area of system design. This gap is a result of two arguments. First, it is a belief that formal approaches are hard to understanding and therefore to use. Second, the formal specification is not suitable for testing because of its non-executable form. Utilization of formal approaches, such as the formalism of OOPN, addresses mentioned disadvantage. Their formal nature combined with graphic representation of models allows them to be used by designers who have minimal knowledge of the formal background. Models described by these formalisms can be simulated as well as integrated into real conditions. All changes in the validation process are entered directly into the model, and it is therefore not necessary to implement or transform models.

To model domain concepts of the system being developed the class diagrams from UML are usually used [7]. Similarly, to specify user requirements the use case diagrams from UML are used. The concept of modeling requirements presented in this paper is based on mentioned UML models and the formalism of OOPN, which is used for behavior specifications. The goal is to combine the advantages of intuitive approach to system modeling with the precise specification of requirements and the detailed description of realization. The concept is demonstrated on a simple case study.

The paper is organized as follows. Section II deals with related work. Section III summarizes the concept of software system modeling and introduces the simple case study. The question of user requirements modeling using use case diagrams is discussed in Section IV. It introduces our extension to use case diagrams by one special relationship. Section V deals with behavior modeling and compares an usage of statecharts from UML and the formalism of OOPN. Modeling use case relationships is discussed in Section VI. Use cases and their behavior described by the formalism of OOPN create the architectural form of modeled system. Mapping use cases, nets, and classes is introduced in Section VII. The summary and future work is described in Section VIII.

II. Related Work

One of the major criticisms of UML is the inability to precisely describe all aspects of the designed system including integrity constraints [8]. The clear understanding, automated transformations, and simulation of models are complicated. One approach to addressing the problem is to introduce the constraints on the model elements. An example is Object Constraint Language (OCL) [9], which allows to precisely specify the semantics of the model elements. Another approach works with modified UML models that can be executed and,

therefore, validated by simulation. An example is the Executable UML (xUML) language [4] used by the Model Driven Architecture (MDA) methodology [10], or Foundational Subset for xUML [11][12] associated with Action Language (Alf) [13].

These methods are faced with a problem of model testing. Models have to be transformed into executable form, whereas the validation of proposed requirements through these models in real conditions is complicated. Transferring changes made during testing back to higher abstraction models is difficult, sometimes impossible. It is a problem because the models become useless over the development time.

Similar work based on ideas of model-driven development deals with gaps between different development phases and focuses on the usage of conceptual models during the simulation model development process—these techniques are called *model continuity* [14][15]. While it works with simulation models during design phases, the approach proposed in this paper focuses on *live models* that can be used in the deployed system.

III. Modeling of Software Systems

To specify the system being designed, a wide range of languages and formalisms can be adopted. The most commonly used means is UML language, which concentrates experiences of other languages used in the past to modeling requirements and behavior of systems. UML, however, fails to capture the essential features of one model. It is necessary to work with different views, but there is no mechanism for easy portability between those views. The basic diagrams are use case diagrams and class diagrams, which are supplemented by other diagrams as necessary. Use case diagrams model the options, how the system can be used, whereas use cases are specified by diagrams of activities or interaction diagrams. Class diagrams are a fundamental domain model and are linked with already mentioned interaction diagrams. The behavior of individual classes can be specified, e.g., by a state diagram.

A. Case Study

We will demonstrate basic principles and problems of user requirements modeling on the simplified example of robotic system. The example works with a robot, which is controlled by the algorithm. Users can handle algorithms for controlling the robot (he/she can choose one algorithm for handling and start or stop the algorithm).

B. Domain modeling

Domain model captures the system concepts, as they are identified and understood during the process of requirements analysis. The domain concepts are modeled by class diagram containing conceptual classes and their relationships. The domain model is the initial model for modeling the functional requirements and creation of design models. It is one of the first models when creating software.

Initial analysis of presented case study suggests that we must be able to work with concepts *User*, *Algorithm*, and *Robot*. Thus, we can create the initial domain model, which is shown in Fig. 1. There is a one-to-N association (1..N) between

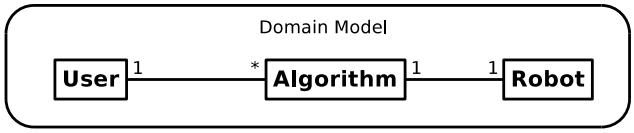

Figure 1. Domain model of the case study.

User and *Algorithm*, since a user may work with multiple algorithms. One particular algorithm controls only one specific robot, so there is a one-to-one association between *Algorithm* and *Robot*.

C. User Requirements Modeling

The use case diagrams are used for modeling of user requirements. The aim is to identify users of the system, the system requirements and how the user can work with requests. The basic elements are therefore *users*, their *role*, and *activities*. Roles are modeled as *actors* and activities are modeled by individual *use cases*. The use case diagram of our example is shown in Fig. 2.

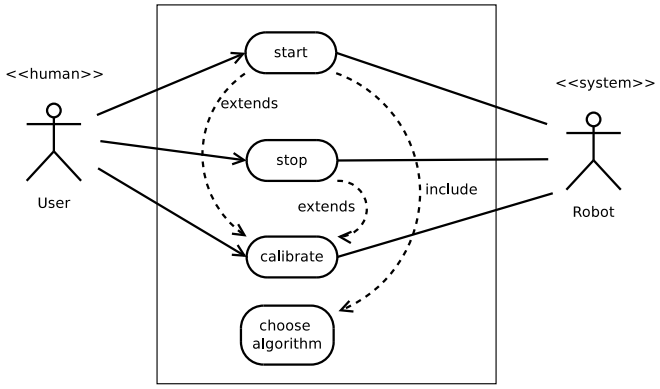

Figure 2. First Use Case Diagram for the robotic system.

The diagram shows actors (roles) *User* and *Robot* and use cases *Start* algorithm, *Stop* algorithm, *Choose* algorithm, and *Calibrate* the system. Roles represent the interface between the system and its surrounding and define operations allowed for that role.

D. Behavior Modeling

Behavior models deal with functional requirements. They model *scenarios*, i.e., specific behaviors and interactions of individual use cases. For that purpose, various types of description are used—structured text, activity diagrams, state diagrams etc. Generally, they are models enabling to capture work-flow supplemented by communications. Scenarios of individual cases are modeled by activity diagrams, state diagrams, or interaction diagrams. However, the formal models and formal languages, such as *Petri nets*, can be used as well. An important feature is the interconnection of use case diagrams and scenarios modeled using specific diagrams, since both types of models represent different view of the developed system.

IV. Modeling User Requirements

This section detail examines the concept of use cases in system design. Use case diagrams (UCDs) are used in the process of software system design for modeling user requirements. The system is considered as a black-box, where only external features are taken into account. The objective of UCDs is identify system users, user requirements, and how the user interacts with the system. The model consists of *actors* and *use cases*. Actor generates an external stimulus of the system and, generally, it represents a kind of users working with the system. Use case models a sequence of interactions between actors and software system. For a description of the interactions, plain text is usually used. The text describes inputs from actors and reactions of the system. Use case defines *what* the system is to do and pays no attention to a question *how* the system would implement modeled requirements.

A. Actor

Actor is an external entity working with the software system, so that actor is not part of the system, but it is a generator of input stimulus and data for the system. Actor models a group of real users, whereas all members of the group are working with the system in the same way. Therefore, actor represents *a role* of the user in which can appear in the system. One real user can appear in the system in more roles. Let us consider the example of conference system with actors *Author* and *Reviewer*. These actors model two roles, each of them defines a set of functions (use cases) the user can initiate or can participate on. The real user can either be author or reviewer, or can work with the system in both roles (the user usually stands in just one role at the time).

Now, let us consider another example of the garage gate handling system. The system consists of actuators (garage gate), sensors (driving sensor, card scanner), and control software. It is closed autonomous system with which two groups of real users can work—*Driver* and *Reception clerk*. The driver comes to the garage gate, applies a card to the scanner, and the system opens the gate. If the user does not have a card, he can ask reception clerk, who opens the gate. From system point of view, actuators, sensors, and control software are internal parts of the system. From the software engineering point of view, actuators and sensors are *external* elements that are controlled by the system, or from which it receives information.

We can ask a question whether we can model these external elements using the actor concept. Actors represent human users in many information systems (*human actors*). But, they can also be used to model other subsystems such as sensors or devices (*system actors*) because of they really represent external entity. The system has to communicate with these subsystems, nevertheless, they need not to be part of the modeled software system. There are systems where this form of actors is more important than users [16]. They concern especially embedded or autonomous systems that intensively cooperate with input–output devices, such as sensors, actuators, etc. These actors represent the surroundings in which the system operates.

It will be useful to define specific categories of actors based on their merit, whose semantics differ from conventional apprehension of the term *actor* in use case diagrams. The categorization follows:

- *real user* (stereotype *human*) – models a real user, or more precisely his/her role in the system, which is concerned in interactions with the system
- sensor, actuator, *device* in general (stereotype *device*) – model a system element, which provides stimulus to control software or receives commands
- *system* (stereotype *system*) – other system (or subsystem) with which the modeled system cooperates

B. Use Case

An important part of functional requirements analysis is to identify sequences of interaction between actors and modeled system. Each such a sequence covers different functional requirement on the system. The sequence of interactions is modeled by *use cases*. The use case describes a main sequence of interactions and is invoked (its execution starts) by input stimulus from the *actor*. The main sequence can be supplemented by alternative sequences describing less commonly used interactions. Their invocation depends on specified conditions, e.g., wrong information input or abnormal system state. Each sequence (the main or alternative one) is called *scenario*. Scenario is a complete implementation of one specific sequence of interactions within the use case.

C. Relationships Between Use Cases

Among the different use cases you can use two defined relationships, *include* and *extend*. The aim of these relations is to maximize extensibility and reusability of use cases if the model becomes too complex. A secondary effect of using of these relationships is to emphasize the dependence of the individual use case scenarios, structuring too long scenarios to more lower level use cases, or highlighting selected activities.

1) Relationship extend: Relationship *extend* reflects alternative scenarios for basic use case. In cases where the specification of a use case is too complicated and contains many different scenarios, it is possible to model a chosen alternative for new use case, which is called *extension use case*. This use case then extends the basic use case that defines a location (point of extension) in the sequence of interactions and conditions under which the extension use case is invoked. The relationship *extend* is illustrated in Fig. 2. The use case *calibrate* has to stop the running algorithm first, then to calibrate the system and, finally, to start it. Use cases *start* and *stop* can thus expand the base case scenario *calibrate*.

2) Relationship include: Relationship *include* reflects the scenarios that can be shared by more than one use case. Common sequence can be extracted from the original use cases and modeled by a new use case, which we will call *inclusion use case*. Such use case can then be used in various basic use cases that determine the location (point of insertion) in the sequence of interactions for inclusion. The relationship *include* is illustrated in Fig. 2. Now, we adjust the original sequence of interactions with the use case *start*, which will need to select the algorithm to be executed first. Use case *start* thus includes the use case *choose algorithm*.

3) Generalization use cases: The activities related to interactions between the software system and a robot were not highlighted yet. One possibility is to define *inclusion use case* describing these interactions, i.e., the algorithm. However, this method supposes only one algorithm, which contradicts the specified option to choose algorithm. Second possibility is to define *extension use cases*, everyone for various algorithms. The disadvantage of this solution is its ambiguity; there is no obvious the problem and the appropriate solution.

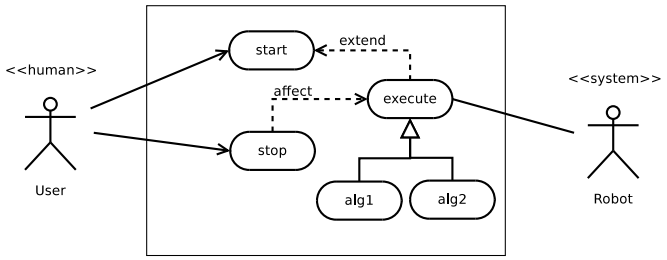

Figure 3. Specialization of the use case *execute* and the relationship affect.

Use case diagram offers the possibility to generalize cases. This feature is similar to the generalization (inheritance) in an object-oriented environment. In the context of the use case diagrams, generalization primarily reflects the interchangeability of the base-case for derived cases. Although there are methods that consider generalization as abstruse [17] and recommend replacing it with relation *extend*, generalization has a unique importance in interpreting the use case diagram. Relation *extend* allows to invoke more extension use cases, whereas generalization clearly expresses the idea that case *start* works with one of cases *execute* (the model is shown in Fig. 3). The model can also be easily extended without having to modify already existing cases.

4) Use Case Diagram Extension: The present example shows one situation that is not captured in the diagram and use case diagrams do not provide resources for its proper modeling. This is the case *stop*, which affects the use case *execute* (or possibly derived cases), but does not form its basis (the case *execute* is neither part of it nor its extension). Nevertheless, its execution affects the sequence of interactions, which is modeled by use case *execute* (it stops its activity). In the classical chart this situation would only be described in the specification of individual cases, however, we introduce a simple extension *affect*, as shown in Fig. 3. Relation *affect* represents a situation, where the base use case execution has a direct impact on other, dependent use case. This relation is useful to model synchronization between cases in such a system, which suppose autonomous activities modeled by use cases.

D. Problems associated with modeling relationships

The disadvantage of the use of relationships between use cases is the ability to determine the nature of addiction without detailed knowledge of the specification. If we analyze the model in Fig. 2, we find that the relations *extend* are not used correctly and can lead to more complications in the design model. The example assumes that when you start the algorithm, the user must always choose the algorithm, which is in connection with the case *calibrate* inappropriate (the algorithm is already selected, just for a moment it was suspended). Moreover, it is not a user interaction, it is therefore preferable to model the suspension and starting the algorithm directly as the activity of the case *calibrate* without using the relation *extend*.

V. Behavior Modeling

Use case specification format is not prescribed and can have a variety of expressive and modeling means, e.g., plain text, structured text, or any of the models. UML offers, among others, the activity and state diagrams. These charts allow precise description based on modeling elements with clear semantics. In this section, we will outline UML based way how to specify the use case *alg1* (one of the possible algorithms for controlling the robot; see Fig. 3).

A. State Diagram

Let us walk through an example of use case *alg1* specification using state diagram. We will discuss only part of the model shown in Fig. 4.

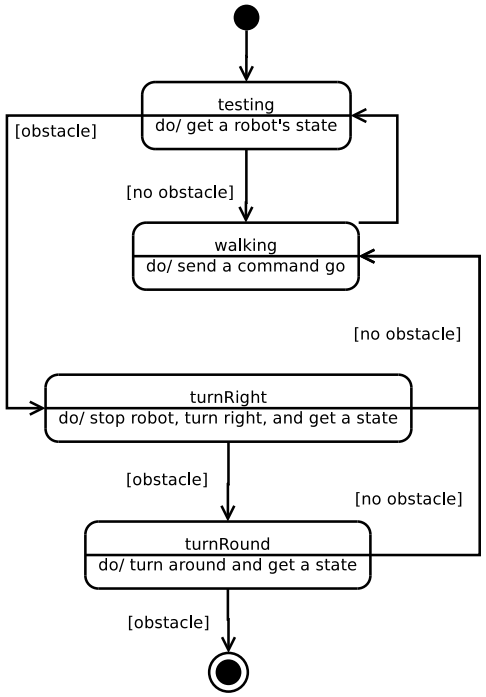

Figure 4. Statechart modeling the use case *Algorithm1* (*alg1*).

The model captures system sub-states, system activities carried out in those states and transitions between states. Execution of transition is conditioned, conditions include activities that are carried out to change the system state (in this example it is not used). States are modeled as elements that can contain internal activities performed by the system or a particular object, which is in this state. Simultaneously, it declares response to external events, i.e., its method of operation depending on the system state. The transition is modeled by edge, whose execution may depend on a condition or external events. An example might be a possibility to stop

the algorithm in any state. We will now analyze the model. When activated, the system will move from the initial state (bulging black full circle) into a state called *testing*. Activity performed in this state is the acquisition of the robot state, i.e., testing that the robot is facing an obstacle. Based on the information obtained, the system moves into one of the states *walking* (the road is clear and the condition *no obstacle* is met) or *turnRight* (the road is not clear, the condition *obstacle* is met).

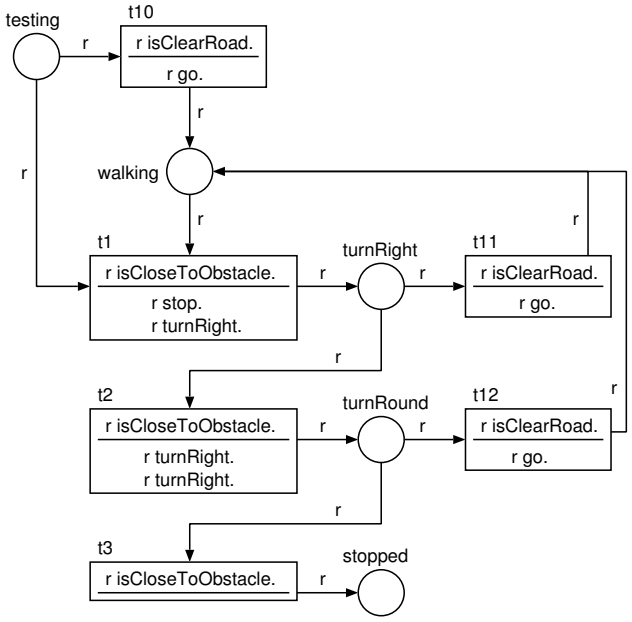

Figure 5. Petri net modeling the use case *Algorithm1* (*alg1*).

These charts allow to describe functional requirements of use case diagrams but their validation is problematic because of impossibility to check models either by formal means or by simulation. Of course, there are tools and methods [4][18] that allow to simulate modified UML diagrams. Nevertheless, there is still a strict border between *design* and *implementation* phases. Another way is to use some of the formal models. In this section, we introduce Object Oriented Petri Nets (OOPN) for specifying *use case*, i.e., interactions between the system and the actors. Let us walk through the previous example of use case *alg1* shown in Fig. 5.

B. Object Oriented Petri Nets

By comparison OOPN model (see Fig. 5) and the state diagram (see Fig. 4) we find a fundamental difference in the way of the states and transitions declaration. The system state is represented by places of the OOPN formalism. System is in a particular state if an appropriate place contains a *token*. Actions taken in a particular state is 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 the tokens between places. Except the input places, the transition firing is conditioned by a *guard*. The guard contains conditions or synchronous ports. The transition can be fired only if the guard is evaluated as true. If the transition fires, it executes the guard, which can have a side effect, e.g., the executed synchronous port can change a state of the other case.

The model (see Fig. 5) consists of states *testing*, *walking*, and *turnRight* that are represented by places. State *turnRight* is only temporal and the activity goes through these ones to the one of stable states (e.g., *walking*). Control flow is modeled by the sequence of transitions, where 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 (modeled by the synchronous port) *isCloseToObstacle* is met. When firing this transition, actions to stop the robot (*stop*) and to turn right (*turnRight*) are performed and the system moves to the state of *turnRight*. The transition *t10* is fireable, if the condition (synchronous port) *isClearRoad* is met. When firing this transition, the action to go straight (*go*) is performed and the system moves into the state *walking*.

Both testing condition and messaging represent the interaction of the system with the robot. The robot moves the control flow as *token*, which allows interaction at the appropriate point of control flow and at the same time defines the state of its location in one of the places. To achieve correct behavior, it is useful to define type constraints on tokens (see $\geq \{Robot\}$; it means the token should be of a type *Robot*). Even as, it clearly shows *which* actor (and derived actors) interacts in those scenarios.

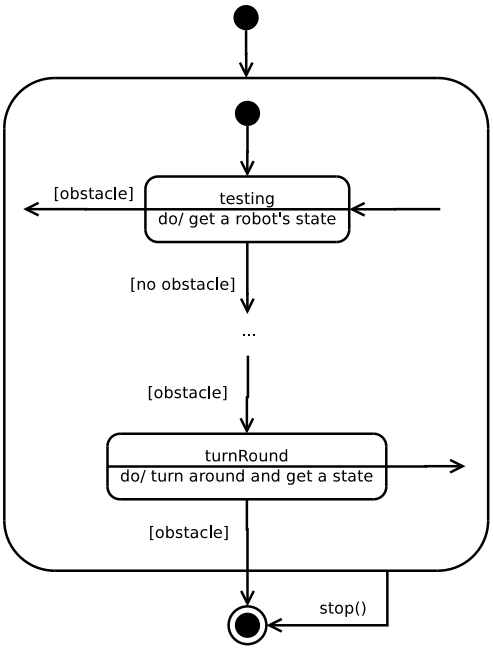

Figure 6. Composite state manipulation in statecharts.

To make decision about moving between states, obtaining data is not separated from state testing. If we look at the state *testing* in Fig. 5, we see that obtaining information and state testing are modeled in the transition guard by calling predicates or synchronous ports over the actor *Robot*.

C. Modeling Alternative Scenarios in State Diagram

Modeling alternative scenarios, i.e., scenarios that supplement the basic scenario, will be described on the case of *stopping algorithm*. It can be modeled by transition that responds to an external event *stop*. Since the transitions of the same type had to be included in every state offers a state diagram for these situations *composite state*, which introduces a certain hierarchy in the modeling phase diagrams. Each folded state is defined by the states and transitions again, there is a possibility to define a transition from a folded condition, which is interpreted as a transition from any state of the folded state. The example is shown in Fig. 6.

D. Modeling Alternative Scenarios in OOPN

Alternative scenarios, i.e., scenarios that supplement the basic scenario, are modeled by synchronous ports (perhaps even methods) to handle a response to an external event. We show a variant of the suspension of the algorithm, i.e., removal of the token from the current state and restoring algorithm, i.e., return the token back to the correct place. We introduce a new state (place) *paused* representing suspended algorithm. Because the formalism of OOPN does not have a mechanism for working with composite states, we should declare auxiliary transitions or ports for each state we want to manipulate with.

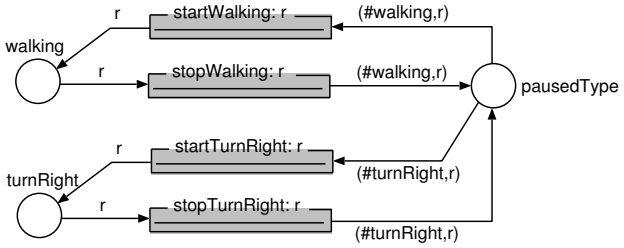

Figure 7. Composite state manipulation in OOPN.

Part of the use case model having established responses to external events *pause* and *resume* is shown in Fig. 7. We have to define an auxiliary place *pausedType* to store information about original placing of the token. For example, the composite synchronous port *pause* is fireable, if at least one of the synchronous ports *stopWalking* and *stopTurnRight* is fireable.

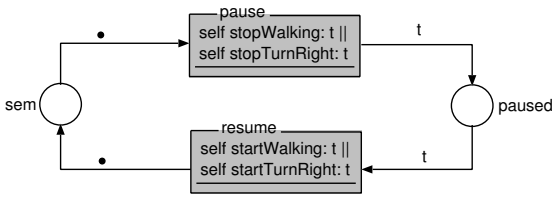

Figure 8. Composite state manipulation in OOPN.

This way of modeling is clear, however, confusing for readability. Furthermore, to work with a larger set of states is almost unusable. Nevertheless, there is the same pattern for each state, so that the concept of collective work with the states is introduced. It wraps the syntax of the original net. This will improve the readability of the model, while preserving

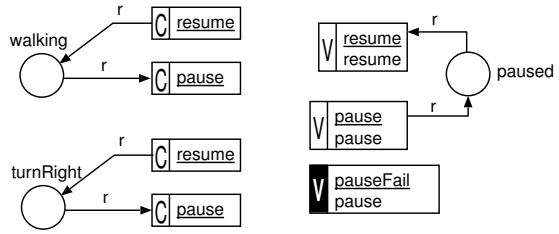

Figure 9. Composite state manipulation in OOPN.

the exactness of modeling by Petri nets including testing models. The example is shown in Fig. 9. The synchronous port is divided into two parts—the *common* part (*C-part*) and the *variable-join* part (*V-part*). The *C-part* represents all synchronous ports, which should be called from the composite port. The *V-part* represents a way how to work with the *C-part*—it is fireable, if at least one item of the *C-part* is fireable.

E. Modeling Roles

Until now we have neglected the essence of the token that provides interaction with the actors and defines the system state by its position. As mentioned, actor represents *role* of the user or device (i.e., a real actor), which can hold in the system. One real actor may hold multiple roles, can thus be modeled by various actors. Actor defines a subset of use cases allowed for such a role. For instance, the *robot* is not allowed to choose algorithm to execute, so its model does not contain any interaction to that use case.

A role is modeled as a use case and it behavior by Petri nets. Interactions between use cases and actors are synchronized through *synchronous ports* that test conditions, convey the necessary data and can initiate an alternative scenario for both sides. Use case can then send instructions through messages too.

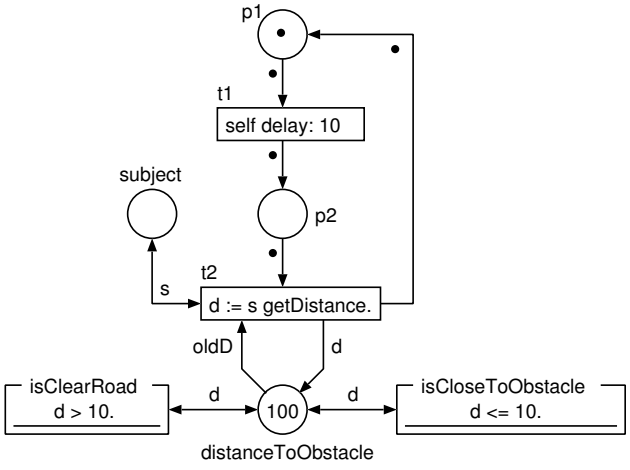

Figure 10. Petri net specification of the role *Robot*.

In our example, we will model the secondary role *Robot*, whose basic model is shown in Fig. 10. Scenarios of the *execute* use cases are synchronized using synchronous

ports *isCloseToObstacle* and *isClearRoad* whose definition is simple—to test the distance to the nearest obstacle, which is stored in the place *distanceToObstacle*. Its content is periodically refreshed with a new value coming through the common place *distance*. The net can define methods for controlling a real actor too.

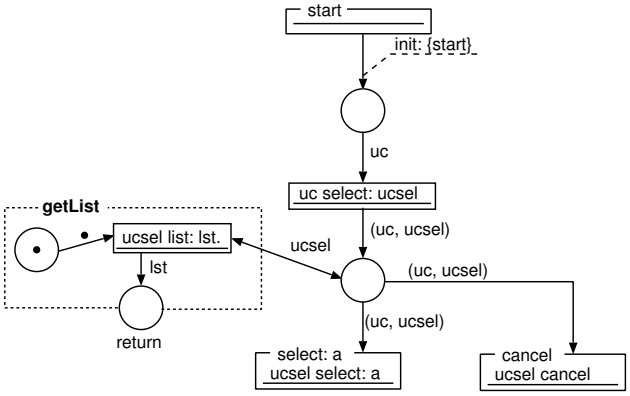

Figure 11. Petri net specification of the role *User*.

Model of the next role *User* is shown in Fig. 11. The primary actor defines stimuli (modeled as synchronous ports and methods) that can perform a real actor. Their execution is always conditioned by an actor workflow and a net of currently synchronized use case. Model shows the workflow of the use case *start*, which starts by calling a synchronous port *start*. It invocates the use case *start* (the syntactically simpler notation is used, it is semantically identical to invocation shown in Fig. 12). Using the method *getList* is possible to obtain a list of algorithms. Allowed actions can be executed by one of the defined synchronous ports *select:* and *cancel*.

VI. Relationships Modeling

We turn now to a method of modeling the relationships between use cases. As we have already defined, we distinguish relations *include*, *extend*, *affect*, and *generalization*.

A. Common Net and Common Places

Modeling the workflow that includes multiple separate synchronized nets may need to share a single network to other networks. For this purpose, the synchronous ports are used. Nevertheless, it can be difficult to read the basic model of the flow of events, because of the need for explicit modeling synchronous ports for data manipulation. Therefore, we introduce the concept of *common net* and *common place*. It is not a new concept, only the syntactic coating certain patterns using synchronous ports.

Each model has defined its initial class that also defines the common net represented by the class *Common* that for each running model has exactly one instance identified by the name *common*. The initial class initiates execution of the simulation and, simultaneously, provides a means for accessing common places. Content of the common places is is available through standard mechanisms (e.g., synchronous ports). Difference to the ordinary usage lies in the fact that access mechanisms are hidden and access to the common places from other nets is modeled by *mapping*—the place marked as common in the other net is mapped onto common place defined in the common net.

B. Modeling the include relationship

We will continue our example and create models of use cases *start* and *choose algorithm*, which is *inclusion case* to the case *start*. Case *start* is activated by actor *user*, connected by a mutual interaction. Actor *user* is the primary actor, so it generates stimulus to that the case has to respond. It implies a method of modeling events in the sequence of interactions. Responses to actor's requirements have to be modeled as an external event, i.e., using a synchronous port. Another significant issue is a place of inclusion into the basic sequence of interactions and invocation activities of the integrated case.

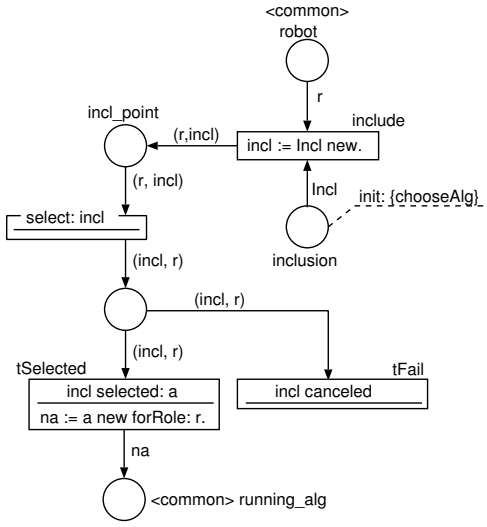

Figure 12. Petri net specification of the use case *start*.

The model of use case *start* is shown in Fig. 12. The inclusion use case is stored in a place *inclusion* and the insertion point is modeled by internal event (transition) *include* with a link to a place *incl_point*. Invoking the use case corresponds to instantiate the appropriate net (see the calling *new* in the transition *include*). The following external event (synchronous port) *select:* initiates the interaction of the actor *user* with integrated activity. The event binds the inclusion case to the free variable *incl*, and simultaneously stores it to an auxiliary place. Conditional branching is modeled by internal activities (transitions) *tSelected* and *tFail*. Their execution is subject to a state of inclusion case, which is tested by synchronous ports in guards. In case of success (transition *tSelected*), the synchronous port *selected:* binds the selected algorithm to the free variable *a* and stores it to the common place *running_alg*.

The use case *choose algorithm* specification is shown in Fig. 13. The basic sequence (to obtain algorithm list and select one of them) is supplemented with an alternative sequence (the user does not select any algorithm) and a condition (empty list corresponds to the situation when a user selects no algorithm). Inclusion case is viewed from stimuli generation point of view as secondary element; its activities are synchronized by basic case or actor, which works to the base case. Synchronization

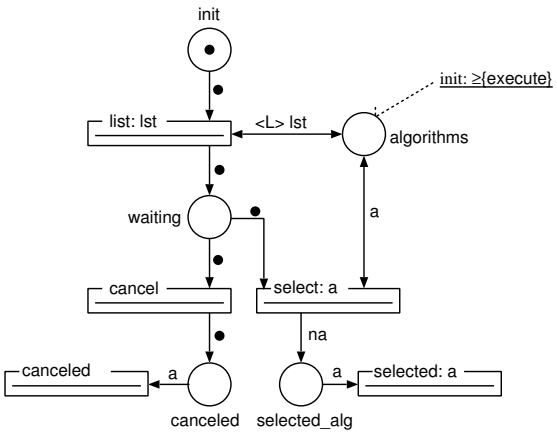

Figure 13. Petri net specification of the use case *choose algorithm*.

points are therefore modeled as external events, i.e., using synchronous ports. The case does not work with any secondary actor, so that to define the status of the net is sufficient type-free token (modeled as dot). The first external event is to obtain a list of algorithms (synchronous port *list:*); the variable *lst* binds the entire content of the place *algorithms*. This place is initialized by a set of cases (nets) derived from the case (net) *execute*. Now, the case waits for actor decision, which may be two. A user selects either no algorithm (external event *cancel*), or select a specific algorithm from the list, which has to match the algorithm from the place *algorithms* (external event *select:*). Token location into one of the places *canceled* or *selected_alg* represents possible states after a sequence of interactions. These conditions can be tested by synchronous ports *unselected* and *selected:*.

C. Modeling the extend relationship

Relation *extend* exists between cases *start* and *execute*, where *execute* is the extension use case. This relationship expresses the possibility of execution of the algorithm, provided that some algorithm was chosen. Since this is an alternative, it is expressed by branches beginning transition *tSelected*, as we can see in Fig. 12. The transition *tSelected* represents the insertion point of the extension of the basic sequence of interactions.

D. Modeling the affect relationship

Relationship *affect* exists between cases *stop* and *execute*, where *stop* influences the sequence of interactions of the case *execute*, respectively any inherited cases. Petri nets model for this use case is shown in Fig. 14. The activity begins from the common place *running_alg* and branches in three variants (transitions *t1*, *t2*, and *t3*). Branch *t1* says *no algorithm is running*; common place *running_alg* is empty. Because OOPN do not have inhibitors, the negative predicate *empty* is used to test conditions, which is feasible, if it is impossible to bind any object to the variable *a*.

Branch *t2* says *the algorithm is running*; the common place *running_alg* contains an active algorithm. Synchronous port *pause* (see Fig. 9) called on the running algorithm is evaluated

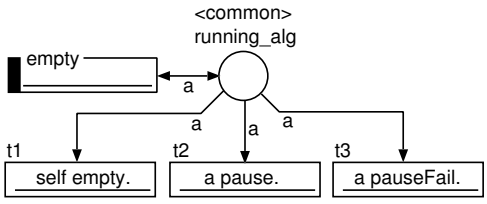

Figure 14. Petri net specification of the use case *stop*.

as true and when performed, it moves the algorithm into *stopped* state. Branch *t3* says *the algorithm is not running*; the common place *running_alg* contains an active algorithm. Synchronous port *pauseFail* called on the running algorithm is evaluated as true and when perform, it has no side effect.

This model is purely declarative. We declare three possible variants that may arise, and simultaneously declare target individual options to be done. Only one variant can be performed at a time. We can define other activities related to these variants. We can see that it does not invoke the use case *execute*, i.e., there is no instantiating a net, but this activity is affected. It is therefore not appropriate to model this situation with the relations *include* or *extend*. After all, it is appropriate to model that relationship.

E. Modeling the generalization relationship

The generalization of use cases does not have the same meaning like the generalization of classes in object-oriented architectural modeling. Special (inherited) case does not develop or modify the basic case, the relationship demonstrates only that fact, that it is possible to use any inherited case instead of the base case. Modeling of the generalization relationship in Petri nets reduces to express the possibility of working at a point defining the relationship *include* or *extend* to some case *c*, with all cases inherited from case *c*. In our example, this situation is shown on the use case model *choose algorithm* (Fig. 13). The place *algorithm* contains all possible algorithms that can be provided, i.e., nets inherited from base use case *execute*. Wherever the case *executed* is used in the model, it is possible to use any inherited case.

VII. Architecture Model

The presented concept of modeling assumes mapping use cases and roles to individual nets. Each net is part of a class, usually defined as the object net and in some cases as the method net. This section introduces the class diagram of the case study, which encapsulates designed nets.

A. Initial Class

Each model has defined its initial class that also defines the *common net* in the form of an object net. The initial class initiates execution of the simulation and, simultaneously, provides a means for accessing common places.

The example uses two common places *robot* and *running_alg*. The place *robot* stores information about the role of robot the whole system works with. The role is initialized by the creation of initial object, as shown in the initial object

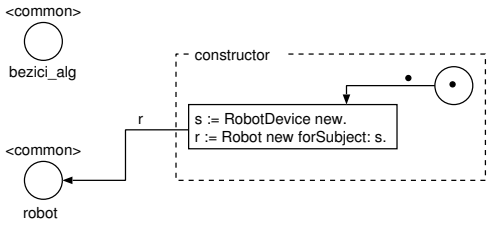

Figure 15. The initial class.

net. As the initialization is done only once, immediately the net is created, this part can be regarded as a constructor (see Fig. 15). The role is working with equipment whose interface is represented by the class *RobotDevice*. At this moment we do not resolve how this class is implemented (Petri nets or domain language). The currently selected algorithm is inserted into place *running_alg* by the activity *Start*.

B. Modeling Real Actors

Real actor can hold many roles that are modeled by actors in the system. Each of these roles always has a common base, which is a representation of the real actor, whether a user, system, or device. The model has to capture this fact. For terminological reasons, in order to remove potential confusion of terms *actor* and *real actor*, we denote a real actor by the term *subject*. The subject is basically an interface to a real form of the actor or to stored data. Therefore, it can be modeled in different ways that can be synchronized with Petri nets. Due to the nature of the used nets, there can be used Petri nets, other kind of formalism, or programming language.

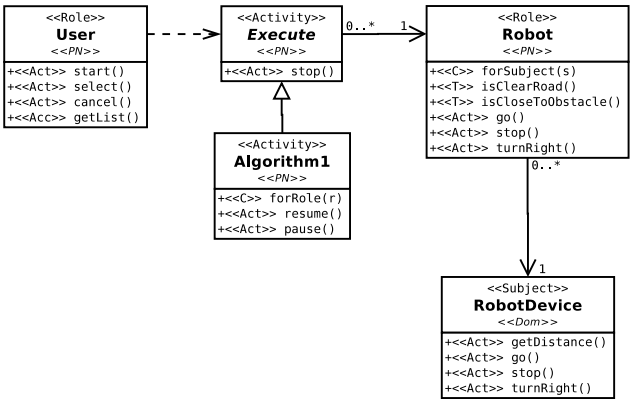

Figure 16. The basic class diagram.

The subject of the role *Robot* is modeled by the class *RobotDevice*. Its operations are implemented by methods in the supported language (in our case it is Smalltalk). This class represents the interface to the real robot, which is controlled by our application. The specific implementation is not important for demonstration of the modeling principles, therefore we do not mention it here.

The subject of the actor *User* can be modeled as a Smalltalk class, whose object can access OOPN objects directly [19][20]. The following pseudo-code shows a simple example of accessing model from the subject implemented in programming language. First, it asks a common net to get a role of user, then invokes synchronous port *start*, a method *getList*, and finally select first algorithm from the list.

```
usr ← common.newUser();
usr.asPort.start();
lst ← usr.getList();
usr.asPort.select(lst.at(1));
```

C. Mapping Petri Nets and Classes

Class diagram of the architecture model is shown in Fig. 16. Classes represent elements of different levels, therefore each class is marked by stereotypes for distinction. Stereotype *Activity* denotes the class representing a use case, the stereotype *Role* denotes the class representing a role and stereotype *Subject* denotes the class representing the subject. Each class can be modeled (described) by different formalism, the next stereotype distinguishes variant formalism. In our example, the formalism of Petri nets (stereotype *PN*) and Smalltalk language (stereotype *Dom*) are used.

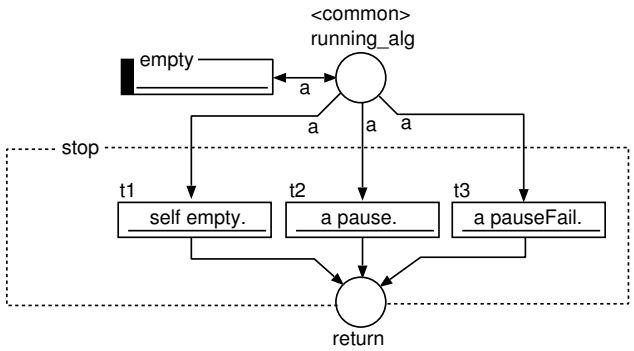

Figure 17. Model of the method net *Execute.stop*.

Class diagram shows the interface of individual elements and the knowledge about other elements. This is indicated by the arrows at the associations. The model shows that *User* is aware of the existence of the activity *Execute* (corresponding with the use case *Execute* in Fig. 3) and its derived activities (classes). The activity *Algorithm1* is created (instantiated) by a user stimulus (this part is not captured). The activity *Algorithm1* is aware of the existence of role *Robot* (it may not be in reverse order), and the role of *Robot* knows about the existence of a subject *RobotDevice* (it may not be in reverse order). The interface operations utilizes stereotypes that correspond to the interface definition. Two kinds of relationships between classes are suggested—the association and the usage. The association expresses a condition where an instance of a class depends on the instance of the second class; dependent instance always contains a reference to the second instance (is part of the place and is often represented by a control token). An example is the association between class *Algorithm1* and the class *Robot*. The usage expresses a condition where the dependent instance may not contain a reference to the second instance, but it can be got through a common place. An example is the usage of the classes *Algorithm1* by the class *User*.

D. Class Execute

The method net *Execute.stop* is shown in Fig. 17. The net is invoked by sending the message *stop*, which corresponds with the use case *Stop*. The method net works with the common place *running_alg* and uses the negative predicate *empty* from the object net.

E. Class Algorithm1

Class *User* represents the role *User*, which enables basic operations with the system *start*, *select*, *cancel*, and *getList*. The operation *start* initiates the use case *Start*.

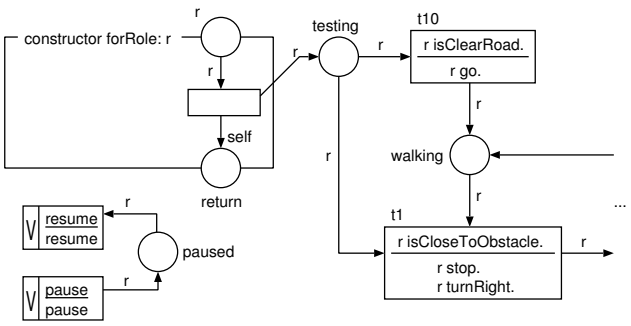

Figure 18. Part of the class *Algorithm1*.

Model of the activity *Algorithm1*, which is represented by the class *Algorithm1*, is shown in Fig. 18. This net communicates with a secondary actor *Robot* through synchronous ports (for synchronization of states) and message passing (for providing commands). The class *Algorithm1* offers the operation *forRole* for activity initialization of the group *constructor* (stereotype *C*) and two operations to stop and resume (*pause* and *resume*) of the group *actions* (stereotype *Act*). If we map classes to OOPN model, we can see that the operation *forRole* is modeled as constructor and both operations *pause* and *resume* is modeled as synchronous ports.

F. Class Robot

The class *Robot* has the constructor *forSubject*, three operations of an *action* group (*go*, *stop* and *turnRight*) and two operations from the *test* group (*isClearRoad* and *isCloseToObstacle*). The operations *stop*, *go*, and *turnRight* can be realized by synchronous ports or methods, depending on the selected communication channel and the internal architecture of the model.

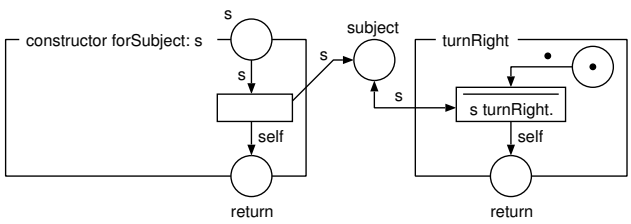

Figure 19. Method nets of the class *Robot*.

Fig. 19 shows the perhaps implementation of the operation *turnRight*, which delegates the execution on the subject. The other operations *go* and *stop* are modeled in a similar way. The subject, with which the role communicates, is stored in the place *subject*. Constructor that initializes the role by inserting the correct subject is shown in Fig. 19.

VIII. Conclusion

The paper presented the concept of modeling software system requirements, which combines commonly used models from UML, such as use case and diagrams, with Petri nets that are not commonly used in the requirements specification. Relationships between actors, use cases, and Petri nets have been introduced. Use case diagrams are used for the initial specification of user requirements while Petri nets serve for use case scenario descriptions allowing to model and validate requirement specifications in real conditions. This approach does not need to transform models or implement requirements in a programming language and prevents the validation process from mistakes caused by model transformations.

At present, we have developed the tool supporting presented approach. In the future, we will focus on the tool completion, a possibility to interconnect model with other formalisms or languages, and feasibility study for different kinds of usage.

Acknowledgment

This work was 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] E. Seidewitz, "UML with meaning: executable modeling in foundational UML and the Alf action language," in HILT '14 Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, 2014, pp. 61–68.

[9] J. Warmer and A. Kleppe, The Object Constraint Language: Getting your models ready for MDA. Longman Publishing, 2003.

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

[11] Object Management Group, "Semantics of a Foundational Subset for Executable UML Models (fUML). OMG Document Number: formal/2013-08-06," http://www.omg.org/spec/FUML/1.1, OMG Document Number: formal/2013-08-06, 2013.

[12] 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.

[13] Object Management Group, "Action Language for Foundational UML (Alf). OMG Document Number: formal/2013-09-01," http://www.omg.org/spec/ALF/1.0.1, OMG Document Number: formal/2013-09-01, 2013.

[14] D. Cetinkaya, A. V. Dai, and M. D. Seck, ACM Transactions on Modeling and Computer Simulation, vol. 25, no. 3, 2015.

[15] X. Hu, "A Simulation-Based Software Development Methodology for Distributed Real-Time Systems," Ph.D. dissertation, The University of Arizona, USA, 2004.

[16] H. Gomaa, Real-Time Software Design for Embedded Systems. Cambridge University Press, 2016.

[17] H. Gomma, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architecture. Addison-Wesley Professional, 2004.

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

[19] 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.

[20] R. Kočí and V. Janoušek, "The Object Oriented Petri Net Component Model," in The Tenth International Conference on Software Engineering Advances. Xpert Publishing Services, 2015, pp. 309–315.