Modeling System Requirements Using Use Cases and Petri Nets

Radek Kočí and Vladimír Janoušek

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

Abstract—The fundamental problem associated with software development is a correct identification, specification and subsequent implementation of the system requirements. To specify requirement, designers often create use case diagrams from Unified Modeling Language (UML). These models are then developed by further UML models. To validate requirements, its executable form has to be obtained or the prototype has to be developed. It can conclude in wrong requirement implementations and incorrect validation process. The approach presented in this work focuses on formal requirement modeling combining the classic models for requirements specification (use case 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 carried 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 paper is part of the System in Simulation Development (SiS) work [1] based on the formalism of Object Oriented Petri Nets (OOPN) [2]. One of the fundamental problems associated with software development is an identification, specification and subsequent implementation of the system requirements [3]. The use case diagram from UML is often used for requirements specification, which is then developed by further UML models [4]. The disadvantage is the inability to validate a specification modeled by that method and it is usually necessary to develop a prototype, which is no longer used after fulfilling its purpose.

Utilization of OOPN enables the simulation (i.e., execute the model), as well as direct integration into a real surroundings, which solves mentioned disadvantage. All changes in the validation process are entered directly into the model, and it is therefore not necessary to implement or transform models. The approach presented in this paper is based on the use cases that are specified by the OOPN formalism. This approach can therefore be mapped to commonly used modeling techniques.

There are methods working with modified UML models that can be executed and, therefore, validated by simulation. An example is the Model Driven Architecture (MDA) methodology [5], language Executable UML (xUML) [6], or Foundational Subset for xUML [7]. These methods are faced with a problem of model transformations. It is complicated to validate proposed requirements through models in real conditions. They either have to implement a prototype or transform models into executable form, which can then be tested and debugged. All changes that result from validation process are hard to transfer back into the models. 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 stages and focuses on the usage of conceptual models during the simulation model development process—these techniques are called model continuity [8]. 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 modeling requirements using use cases from UML and describes our extension to one special relationship. Section III deals with use case specification using OOPN. Modeling use case relationships is discussed in Section IV and the way of actor modeling is described in Section V. The summary and future work is described in Section VI.

II. USE CASE DIAGRAMS

Use case diagrams (UCDs) are used in the process of software system design for modeling functional 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.

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 work with the system by the same way. Therefore, actor represents a role of the user in the system. A real user can play multiple 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.

Now, let us consider the example of a system of garage gate handling. 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 items the system can handle.

So, if actuators and sensors are not internal parts, could we model them using 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). The system has to communicate to such subsystems, nevertheless they need not to be parts of the modeled software system.

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 a use case. 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 conditions. Each sequence (the main or alternative one) is called scenario. Scenario is complete implementation of a 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. 1. 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. 1. 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.

D. 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.

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 [9] 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. 2). The model can also be easily extended without having to modify already existing cases.

E. 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, execute 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. 2. 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.

III. USE CASE SPECIFICATION USING PETRI NETS

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 an activity diagram, a state diagram, etc. These charts allow precise description based on modeling elements with clear semantics, 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 [6][10] 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’s walk through the previous example of use case alg1 shown in Fig. 3.

A. States and Transitions Declaration

The system state is represented by places in 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.

B. Common Net and Common Places

For 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. For each model, we introduce one common net represented by the class CommonNet that for each running model has exactly one instance identified by the name common. The object net of CommonNet may contain common places, i.e., place whose content 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.

C. Modeling of Interaction Sequences

The states testing, walking, and turnRight 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).

![Figure 3. Petri net modeling the use case Algorithm1 (alg1).](image)

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) isConnectedToObstacle 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) isConnectedRoad 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 \( \geq \{ \text{Robot} \}; \)
it means the token should be of a type Robot). Even as, it clearly shows which actor (and derived actors) interacts in those scenarios.

D. Alternative Scenarios Modeling

Alternative scenarios, i.e., scenarios that supplement the basic scenario, is 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 for each state we want to manipulate with.

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 the exactness of modeling by Petri nets including testing models. The example is shown in Fig. 4. 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, that 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.

IV. 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. Modeling of the relation include

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.

The model of use case start is shown in Fig. 5. 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. 6. 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 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:

B. Modeling of the relation extend

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. 5. The transition tSelected represents the insertion point of the extension of the basic sequence of interactions.

C. Modeling of the relation affect

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. 7. 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.

```
empty     <common> running_alg
  ↓       ↓
  a       a

  t1         t2       t3
  self empty pause       a pauseFail.
```

Figure 7. Petri net specification of the use case stop.

Branch t2 says an algorithm is invoked and run; the common place running_alg contains an active algorithm. Synchronous port pause (see Fig. 4) called on the running algorithm is evaluated as true and when performed, it moves the algorithm into stopped state. Branch t3 says an algorithm is invoked and 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.

D. Modeling of the relation generalization

This relationship demonstrates that it is possible to use any inherited case instead of the base case. If there is a point defining the relationship include or extend to a base case c, we can work with any case inherited from the base case c. In our example, this situation is shown on the use case model choose algorithm (Fig. 6). 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.

V. ACTOR SPECIFICATION

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. Modeling Roles

An actor is modeled as a use case, i.e., 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.

```
<common> distance
  ↓
  d

  isClearRoad
  ▼
  d > 10.

<common> request
  ↓
  d

  isCloseToObstacle
  ▼
  d <= 10.
```

Figure 8. Petri net specification of the actor Robot.

In our example, we will model the secondary actor Robot, whose basic model is shown in Fig. 8. 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.

Model of the next actor User is shown in Fig. 9. 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 invokes the use case start (the syntactically simpler notation is used, it is semantically identical to invocation shown in Fig. 5). 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.

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, that 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 (e.g., DEVS), or programming language (Smalltalk until now).

For instance, the subject of the actor Robot can be modeled as an external component, which is linked with the actor through the component interface consisting of one input port distance and one output port request (shown in Fig. 8). These ports are modeled as common place, so that the common net can serve for component interfacing [11]. The subject of the actor User can be modeled as a Smalltalk class, whose object can access OOPN objects directly [12]. The following pseudocode 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.

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

VI. Conclusion

The paper presented the concept of modeling software system requirements, which combines commonly used use case diagrams with not so commonly used Petri nets. The relationship between actors, use cases, and Petri nets has been introduced. Use case diagram is used for the initial specification of functional requirements while Petri nets serve for use case scenario descriptions allowing to model and validate requirement specifications in real surroundings. 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 and languages, and feasibility study for different kinds of usage.

ACKNOWLEDGMENT

This work was supported by the internal BUT project FIT-S-14-2486 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