Modeling and Simulation-Based Design Using Object-Oriented Petri Nets: A Case Study

Radek Kočí and Vladimír Janoušek

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

Abstract. The aim of the paper is to show basic elements of a system design methodology which uses Object oriented Petri nets. The methodology features conformity with UML and uses simulation as a means to verify the models in all system development phases. Simulation also helps in making decisions about structural and behavioral specification of the system. The paper will demonstrate layered modeling technique based on Object oriented Petri nets.

1 Introduction

Modeling and Simulation-Based Design (MSBD) of systems denotes a set of techniques and tools intended for the software system development which is based on formal models, model continuity, and simulation techniques. Its goal is to increase efficiency and reliability of development processes including the software system deployment. The key activities in the system development are specification, testing, validation, and analysis (e.g., of performance, throughput, etc.). Most of the methodologies use models for system specification, i.e., for defining the structure and behavior of developed system. There are different kinds of models, from models of low-level formal basis to pure formal models. Each kind has its advantages and disadvantages. Less formal models (e.g., UML) allows to quickly describe basic system concepts, in the other hand, they do not allows to check the system correctness or validity by means of testing or formal methods—the system has to be implemented before its testing. The more advanced approaches (e.g., Executable UML and Model Driven Architecture [15]) allow to simulate models, i.e., to provide simulation testing. The pure formal models (e.g., Petri Nets, calculus, etc.) allows to use formal or simulation approaches to complete the testing and analysis activities.

The paper aims at system specification using a formalism of Object Oriented Petri Nets [2, 3] (OOPN). The idea of merging Petri nets and objects has been found and elaborated in the 1990's independently by several researchers. The approach closest to our work is the system Renew [11] and associated formalism of Nets-in-Nets [16, 14, 1]. Renew supports modeling of systems using layered Petri Nets and Java language. Similarly to the system Renew, the proposed approach fully supports an integration of formal objects described by Petri Nets and other objects (e.g., it allows to reference and communicate with Smalltalk objects and Petri Net objects uniformly). This feature eases interfacing objects with surrounding world and consequently facilitates hardware-in-the-loop simulation. The idea of using models in all development stages, in conjunction with hardware-in-the-loop simulation, was working up in several projects and is supported by several tools, e.g., the MetaEdit System [13] or Simulink [12]. MetaEdit supports Domain Specific Modeling which allows to generate code from high-level models defined for the domain-specific language. Simulink is aimed at design control systems and hardware architectures. It allows for hardware-in-the-loop simulation for testing designed models in a real environment. Contrary to the works cited above, the proposed approach uses the same model in all development phases including deployment (or final implementation). The formalism of OOPN, in conjunction with the design and simulation framework PNtalk [5], can be directly interpreted and, consequently, integrated into the target system [6].

This paper summarizes methodical approach to system design using Object Oriented Petri Nets. It is a result of previous activities on the field of system design techniques [4, 7], modeling techniques [10], simulation testing and analysis [8], and combination of the OOPN formalism and the UML language [9]. The paper is organized as follows. First, we introduce the used formalism of Object Oriented Petri Nets in section two. The section three describes the basis of design methodology resulted from principles of Modeling and Simulation Based Design. The next three sections deal with particular parts of design methodology including the demonstration on the simple case study. We conclude by summarizing of results and definition of future works.

2 Modeling Formalisms

2.1 Object Oriented Petri Nets

An *object-oriented Petri net* (OOPN) is a triple (Σ, c_0, oid_0) where Σ is a system of classes, c_0 an initial class, and oid_0 the name of an initial object from c_0. A *class* is specified by an object, a set of method nets, a set of synchronous ports and negative predicates. Object nets describe possible autonomous activities of objects, while method nets describe reactions of objects to messages sent to them from the outside. Each net is described by means of high-level Petri nets.

Object nets consist of places and transitions. Every place has its initial marking. Every transition has conditions (i.e., inscribed testing arcs), preconditions (i.e., inscribed input arcs), a guard, an action, and postconditions (i.e., inscribed output arcs). *Method nets* are similar to object nets but, in addition, each of them has a set of parameter places and a return place. Method nets can access places of the appropriate object nets in order to allow running methods to modify states of objects, which they are running in.

Object nets can also contain special kinds of transitions—synchronous ports and negative predicates. *Synchronous ports* are special transitions, which cannot fire alone but only dynamically fused to some other transitions, which activate them from their guards via message sending. Every synchronous port embodies a set of conditions, preconditions, and postconditions over places of the appropriate object net, and further a guard, and a set of parameters. Thus, synchronous ports combine concepts of *transitions* (they have to satisfy preconditions and guards; if the synchronous port is fired, the postconditions are performed) and *method nets* (they have to be called from a guard of another transition).

A synchronous port can be activated via a message sent from a guard of some transition. During transition fireability testing, the searching for suitable variables binding uses backtracking mechanism which takes in account also guard expressions which can consequently test synchronous ports fireability. A synchronous port can be activated with either bound, or unbound formal parameters. In the second case, activation of the synchronous port can bind the formal parameter to some value which can be further used by the calling transition.

Negative predicates are special variants of synchronous ports. Its semantics is inverted—the calling transition is fireable if the negative predicate is not fireable.

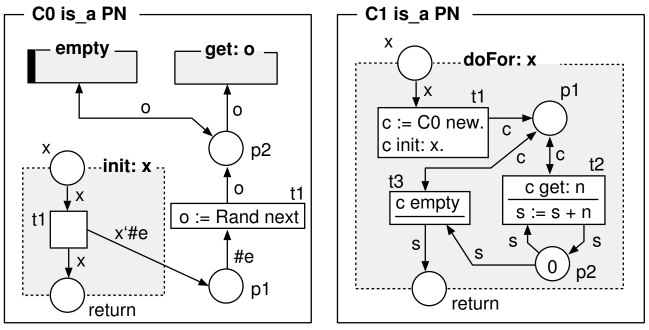

Fig. 1. An OOPN example.

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

Let us investigate what happens if we call the method `doFor:` with a value `3` on the instance of a class `C1` (the instance will be denoted by `obj1`). First, the transition `t1` is fired with following actions: the instance of the class `C0` is created (the instance will be denoted by `obj0`) and the symbol `#e` is put to the place `p1` of the object net `obj0` three times (see the method net `init:`). Now, the object net of `obj0` generates three random numbers (the transition `t1`) and puts them into the place `p2`. Second, the transition `t2` of the object net `obj1` tests if there is any random number in the object net `obj0`—then the synchronous port `get:` is firable. If the transition `t2` fires, the synchronous port `get:` fires too. Since the variable `n` is unbound, the calling binds any random number from the place `p2` of the object net `obj0` to the variable `n`. The transition `t2` of the object net `obj1` then adds this value to the sum (the variable `s`). Third, the transition `t3` of the object net `obj1` tests if there is no random number in the object net `obj0`—then the negative predicate `empty` is firable. If the transition `t3` fires, it places the sum (the variable `s`) to the return place as a method result.

3 Modeling and Simulation Based Design Technique

Modeling and Simulation Based Design (MSBD) is a technique of system design where the system is specified in a form of an executable model which can be verified using simulation experiments. During the development, the model is incrementally refined and each development step is tested and verified. There are two phases which rotate until the system development is finished: Modeling phase and simulation phase. We will especially take into account the techniques of the modeling. In the following sections, we will demonstrate their basic concepts in a small case study.

3.1 Modeling Technique

The modeling technique is focused on the technique of system description, i.e., how the models are created. The technique stems from the classic approach of class identification and definition and extends it to the new features. The design process comprises:

- the identification of use cases of the system,
- the specification of roles and active subjects,
- the specification of activity nets—it is similar to workflow modeling,
- the specification of application nets.

The models are layered hierarchically as shown in Figure 2. Each arrow shows what layers encapsulated another ones. There is a special relationship between use cases in UML and activity nets, and roles in UML and roles nets. These mentioned nets represent appropriate use cases and roles in the system (see the sections 4 and 5). Each role net encapsulate one active subject (see the section 4.1). Each role encapsulate activity nets (see the section 5.3). Moreover, each role can encapsulate another role, and the active subject can also encapsulate another active subject. It allows to get a new view to the role (or active subject) based on the existing one.

The way of system usage is defined by application nets which encapsulate roles nets. Each role has its own set of allowed activities which offers to application nets. The application net can then instantiate and use this activity (see the section 6). The execution of layered nets are synchronized by means of synchronous ports. The nested nets define synchronous port for synchronization of executions and the net at higher layer is controlled by calling these ports. This principle will be demonstrated at the appropriate places in following parts.

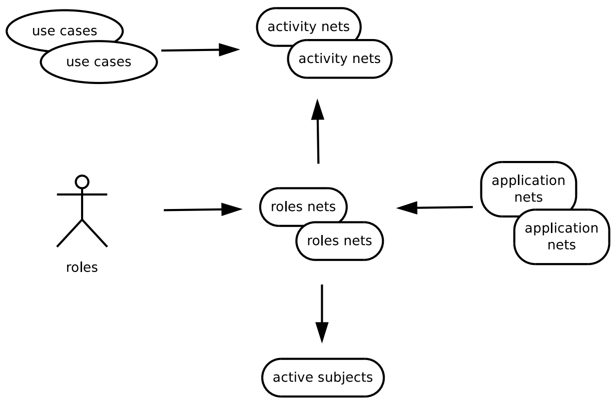

Fig. 2. The layered architecture: an overview.

The design activities are performed in a sequence *use cases–roles and subjects–activity nets–application nets*. But, the system is developed incrementally, in each step, we model selected parts, make decision what part is to be modeled at what layer, and, if necessary, we decide what nets should be changed. Thus, the developer has to go back to designed layer and modify them. The decisions are supported by simulation techniques, i.e., designed models are simulated, the statistic data can be collected, the condition testing can be performed, etc.

3.2 Case Study

We will demonstrate the key features of presented modeling technique on the simple case study. It concerns the reservation system whereas the structure and workflow is only taken into account. The real data associated with the real system will not be modeled.

As we mentioned in the brief description of design process, the process starts with use cases identification. The use case diagram is one of the key diagrams in system specification defined by the Unified Modeling Language (UML). The use case defines a functionality of the system, it is usually complex set of functions to achieve a particular behavior. There is a set of actors who can interact with the use case. The use cases of our case study is shown in Figure 3—there is one actor and two use cases. It represents a system allowing users (an actor *User*) to sign up to the system (a use case *Login*) and to edit its reservations (a use case *Edit Reservation*). In UML, the use case is supplemented with its specification

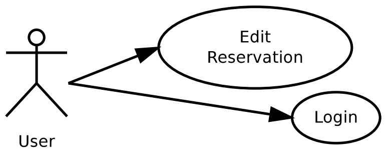

Fig. 3. The use case diagram.

(a text description or other models from UML). Presented technique supposes that the use case is specified by OOPN as will be demonstrated in the section 5.

4 Roles and Active Subjects

4.1 Specification of Active Subjects

Each actor from use case diagrams has its own subset of use cases it can participate with. However, different actors usually have the same basis, e.g., the user represented by its name can act as different actors (administrator, customer, manager, etc.) having different set of behavior (use cases). So we can identify *active subjects* (e.g., person) and their *roles* (e.g., customer, manager). Although we call it an active subject, it does not perform any autonomous action. It only models current state and possible actions shared by all its roles.

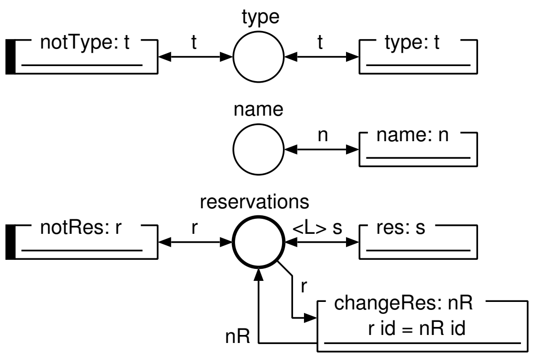

Fig. 4. The active subject Member (modeled as OOPN class).

The example of an active subject is shown in Figure 4. It depicts the object net of OOPN class `Member` storing information about person's *name*, *type*, and set of *reservations*. The type is a set of role identifications the member can act as. Attributes are stored in places (`name`, `type`, and `reservations`) and are accessible by means of synchronous ports. Ports including negative predicates can also serve for testing whether some value of the attribute is set or not.

In real system, the reservations should be stored in some database system, but it is possible to use places for the same purpose for a relative small number of records. But, in this case, there is a problem how to model the iteration of the place content effectively, e.g., if we want to show list of records. The arc between the place `reservations` and the synchronous port `res:` is denoted by a symbol `<L>`. It means that the variable `s` is bound to whole content of the place and this content is accessible as a list (the list is assigned to the variable `s`). Then it is possible to use conventional approach to this list. To change a reservation, the synchronous port `changeRes:` is defined. Each reservation has its unique identification accessible via a call `id`. The event is fireable, if there is the reservation (`r`) in the place `reservations` with the same identification as the given one (`nR`). Then the old reservation is replaced by its new variant.

4.2 Specification of Roles

The member can act in different roles—for our needs we will describe only one role called *User*. The role is modeled as an object net of OOPN class `User`. The example is shown in Figure 5. The role should know about an active subject this role is intended for. In our example, this information is stored in the place `member`.

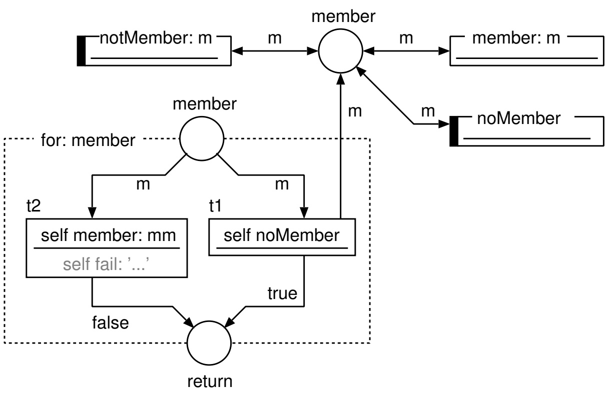

Fig. 5. The role User (modeled as OOPN class).

The net `User` defines two testing negative predicates `notMember:` (it is true if the role does not represent given member `m`) and `noMember` (it is true if there is no represented member) and one synchronous port `member:`. The synchronous port can server for testing (if the role represents given member `m`) or for attribute collection (the example will be shown in chapter 5).

The attributes should be initialized by method nets. Figure 5 shows one net as an example. The method net `for:` initializes the attribute *member* and tests if the role was not initialized yet. If the role is not initialized, the transition `t1` is fireable (the negative predicates `noMember` is true). If the role is already initialized, the transition `t2` is fireable (the synchronous port `member:` is true for a member `m`). In this case, the method net returns `false` and can generate an exception (the calling of `self fail: '...'`); it is useful for testing.

4.3 System as a Special Role

The system usually needs means for persistence, accessing shared objects etc. For this purposes, we introduce a special role net called `Application` (see Figure 6). It allows for getting members (the external event `member:`), storing logged users (the external event `newUser:`), etc.

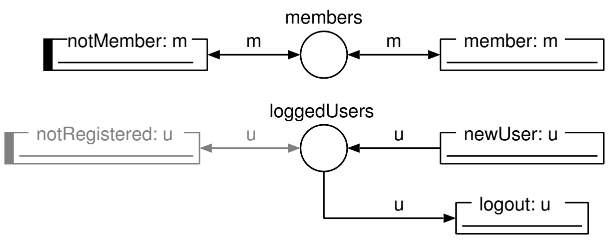

Fig. 6. The role Application (modeled as OOPN class).

5 Activity nets

5.1 Specification of Activity nets

An activity net describes a use case of the system. Each use case is modeled as an OOPN class. Its object net contains transitions, synchronous ports, and places. Since a transition is conditioned only by its input places, it models *internal event* in the activity. On the other hand, a synchronous port is intended for activation from the outside of the activity object. Therefore the synchronous port represent *external event*. Each place in the activity net represents the state of the activity. The state can be tested by means of synchronous ports or negative predicates.

Let us demonstrate this principles in our small example. The example defines `Member`'s role `User` who participates in the use case *editReservation*. The activity net which corresponds to the use case is modeled by the object net of OOPN class `EditReservation` which is shown in Figure 7. The activity net has to know about the role which is associated with the activity. The role is stored in the place `user`. This attribute should be initialized by a method net—because the concept is similar to the net intended for the same purpose (e.g., the net `for:` in the role `User`), the implementation is not shown here.

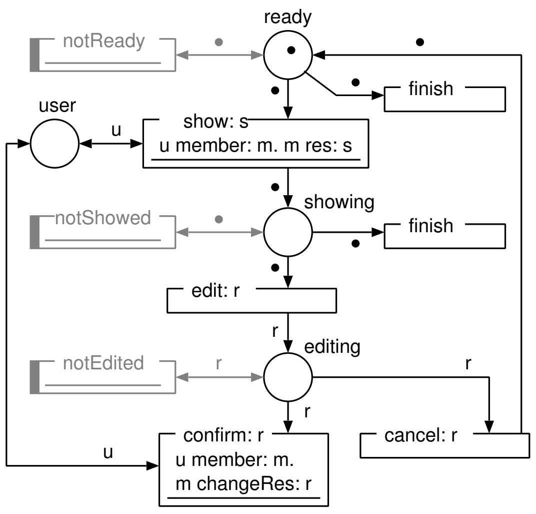

Fig. 7. The activity EditReservation (modeled as OOPN class).

The basic workflow consists of following events: list of reservations representation (see the external event `show:`), one reservation editing (the external event `edit:`), and confirmation (the external event `confirm:`) or to cancellation (the external event `cancel:`) of changes. The net defines three places representing three states of the activity (*ready*, *showing*, and *editing*). The activity can be finished from states *ready* and *showing* by external event `finish`.

It is possible to add synchronous ports for testing activity states. There are defined events for testing whether the activity is not in the defined state, modeled as negative predicates `notReady`, `notShowed`, and `notEdited`. This predicates are true (fireable) only if the state is not satisfied.

5.2 Specification of Actions

The activity usually needs to define some actions. Actions are associated either with internal events or with external events. In the case of internal events, actions are defined inside the transitions (in the action part) or in subnets—then the net is modeled as a method of the appropriate OOPN class and is called from the internal event. In the case of external event, actions are modeled by calling another external events (synchronous ports) from the event's guard.

Figure 7 shows the second approach. For example, the external event `show:` detects the member which is represented by the role `User` (calling the synchronous port `u member: m`—because the variable `m` is unbound, the object net `Member` stored in the place `member` in the object net `User` is bound to the variable `m`). Then the synchronous port `m res: s` is called and the variable `s` then the bound to a list of reservations. The transition, which calls this event, can then use this list (it is shown in the chapter 6).

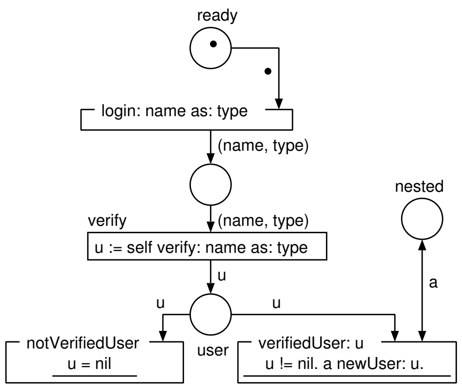

Fig. 8. The activity Login.

Figure 8 shows the activity net *Login* corresponding with the use case *Login*. The basic workflow consists of following events: user's data getting (the external event `login:as:`), the data verification (the internal event `verify`), and testing if the desired role has been created (the external event `verifiedUser:`) or not (the external event `notVerifiedUser`). The action of user verification is associated with the internal event modeled as the sub-net `verify:as:` called from the internal event `verify`. The sub-net is not shown because its implementation is not important for this paper.

There are also actions associated with the external event. If the user is verified and its role is detected (the external event `verifiedUser:`), one action is performed—adding the role into the special role `Application` (`a newUser: u`); see the section 4.3.

5.3 Instantiation of Activity Nets

The activity nets have to be instantiated to serve for their purpose. Because each activity is usually allowed for only specified roles, it is just a role which can instantiate an activity. The example for the role `User` is shown in Figure 9 on the left. The role allows for reservation editing so that the role defines the method net `newEditReservation` which creates a new activity net for reservation edit.

Each activity has to know for which role it is created—it is set by the message `for:`.

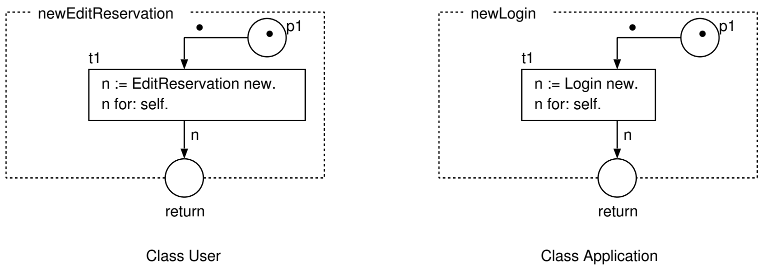

Fig. 9. The methods creating instances of activities.

There is a special kind of activity nets associated with the system instead of the role. These activities are instantiated from special roles as they was introduced in the section 4.3. As an example we can take an activity of *user login* associated with the role `Application`. Figure 9 on the right shows the instantiation of the activity net `Login` from the role `Application`; the principle is same as in ordinary roles.

6 Application Nets

Application nets model a presentation layer of the system. We can also imagine it as an interface to the system. The application net uses roles to generate a list of permitted activities and to instantiate the concrete activity net. The particular behavior of application nets is then controlled by the appropriate activity net. The application nets execution is synchronized by means of external events (synchronous ports) of control (activity) nets.

The presented case study needs some kind of user interface—the part of application net `UserInterace` is shown in Figure 10. The application net `UserInterface` represents an interface to behavior of one role of `User`—how the user can work with the system. The net knows the user (i.e., the role `User`) and the special role `Application`.

Let us investigate the modeled functionality of the application net. First, the transition `login` tests if the user is logged to the system (see the negative predicate `noUser`). If there is no logged user (no role `User` is stored in the place `user`, the activity net `Login` is created (via the special role `Application`) and then the method net `loginUser:` is called. Its execution is controlled by the activity net `Login` using synchronous ports and negative predicates. If the login action is successful, the created role `User` is placed into place `user`. If the login action failed, the procedure is repeated, i.e., the transition `login` is fired again.

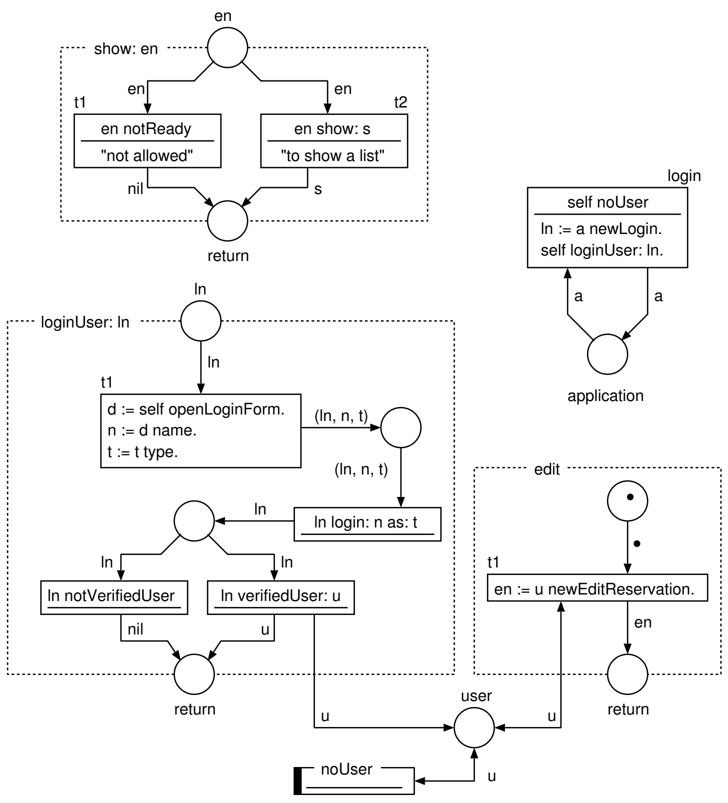

Fig. 10. UserInterface. The topmost level of the application modeled as OON class.

The procedure can be performed at most one at the same time. It is assured by the precondition of the transition `login` to the place `application`.

Second, if the logged user perform an action *edit reservation*, the activity net `EditReservation` is created (see the transition `t1` in the method net `edit`). The activity net then serves as a control mechanism watching if some operation is allowed or not. For example, the method net `show:` makes decision if it is possible to show the reservation list (the activity net is in appropriate state) on not. The method net has `show:` one parameter `en` of the activity net which is used for control.

The model of application net uses rather asynchronous processing of events. For instance, the methods `edit` and `show` represent fragments of a behavior controlled by the activity net `EditReservation`. Each fragment is called as a reaction to the event. For example, there can be generated a graphic user interface offering a button to start reservation editing—if the actor press this button, it generates an event and the method net `edit` is called and the instance of activity net `EditReservation` is created. Then, the first event of activity is to show a list of reservation—as a reaction, the method net `show:` is called. Its execution checks the activity state and collects a list of reservations (calling the synchronous port `en show: s` in the transition `t2`). Then the list can be displayed. The details of graphics user interface and its cooperation with application nets are not described here.

7 Conclusion

The paper dealt with Modeling and Simulation Based Design using the formalism of Object Oriented Petri Nets. It presented the key ideas of a modeling technique which is based on layered approchach. The presented approach is a part of the development methodology, which allows to use formal models in all phases of system development including as basic design, analysis and also programming means with a vision to allow to combine simulated and real components and to deploy models as the target system with no code generation.

So far, the models are interpreted in deployed application and are connected to other software components. The advantage is a possibility to monitor, to profile, to test, and to debug application at the model level with no needs to model transformations. The disadvantage is a possible inefficiency of the model interpretation. Therefore we plan to investigate an approach allowing to transform models into low-level models. Nevertheless, this transformation have to fulfil a condition of having a view to the system at the model level. It means, that it has to be possible to get a state from low-level models and to impose a new state to the low-level model.

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

References

1. L. Cabac, M. Duvigneau, D. Moldt, and H. Rölke. Modeling dynamic architectures using nets-within-nets. In *Applications and Theory of Petri Nets 2005. 26th International Conference, ICATPN 2005, Miami, USA*, pages 148–167, 2005.
2. M. Češka, V. Janoušek, and T. Vojnar. *PNtalk — a Computerized Tool for Object Oriented Petri Nets Modelling*, volume 1333 of *Lecture Notes in Computer Science*, pages 591–610. Springer Verlag, 1997.
3. V. Janoušek and R. Kočí. PNtalk Project: Current Research Direction. In *Simulation Almanac 2005*. FEL ČVUT, Praha, CZ, 2005.
4. R. Kočí and V. Janoušek. System Design with Object Oriented Petri Nets Formalism. In *The Third International Conference on Software Engineering Advances Proceedings ICSEA 2008*, pages 421–426. IEEE Computer Society, 2008.
5. R. Kočí and V. Janoušek. On the Dynamic Features of PNtalk. In *International Workshop on Petri Nets and Software Engineering 2009*, pages 189–206. University of Pierre and Marie Curie, 2009.

6. R. Kočí and V. Janoušek. *Simulation Based Design of Control Systems Using DEVS and Petri Nets*, volume 5717 of *Lecture Notes in Computer Science*, pages 849–856. Springer Verlag, 2009.
7. R. Kočí and V. Janoušek. Towards Simulation-Based Design of the Software Systems. In *The Fourth International Conference on Software Engineering Advances – ICSEA'09*, pages 452–457. IEEE Computer Society, 2009.
8. R. Kočí and V. Janoušek. OOPN and DEVS Formalisms for System Specification and Analysis. In *The Fifth International Conference on Software Engineering Advances*, pages 305–310. IEEE Computer Society, 2010.
9. R. Kočí and V. Janoušek. Towards Design Method Based on Formalisms of Petri Nets, DEVS, and UML. In *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, pages 299–304, 2011.
10. R. Kočí, V. Janoušek, and F. Zbořil. Object Oriented Petri Nets - Modelling Techniques Case Study. *International Journal of Simulation Systems, Science & Technology*, 10(3):32–44, 2010.
11. O. Kummer, F. Wienberg, and et al. *An Extensible Editor and Simulation Engine for Petri Nets: Renew*, volume 3099 of *Lecture Notes in Computer Science*, pages 484–493. Springer Verlag, 2004.
12. MathWorks. Simulink – Simulation and Model-Based Design. http://www.mathworks.com, February 2010.
13. MetaCase. Domain-Specific Modeling with MetaEdit+. http://www.metacase.com, February 2010.
14. D. Moldt. OOA and Petri Nets for System Specification. In *Object-Oriented Programming and Models of Concurrency*. Italy, 1995.
15. C. Raistrick, P. Francis, J. Wright, C Carter, and I. Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.
16. R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, volume 120. Springer-Verlag, 1998.