SIMULATION AND DESIGN OF SYSTEMS WITH OBJECT ORIENTED PETRI NETS

Radek Kočí [1] **and Vladimír Janoušek** [1]

[1]Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 66, Brno, Czech Republic

koci@fit.vutbr.cz (Radek Kočí)

Abstract

Software engineering is a science discipline dealing with methods and techniques of the system design. Increasing complexity of developed systems makes the design process more exacting. The need for better quality of the development processes is growing up too. As an answer to these requirements, new software engineering methods are raising. They are commonly known as *Model-Driven Software Development* or *Model-Based Design* (MBD). An important feature of these methods is the fact that they use executable models, for instance, the most popular one is *Object Management Group's Model Driven Architecture* (MDA) based on Executable UML. The designer creates models and checks their correctness by simulation so that there is no need to make a prototype. The development methods allow for semi-automatic translation of checked models to implementation language (i.e. the code generation). Unfortunately, the resulting code is not final, the code is supposed to be adapted and these changes are usually not moved back to models. Consequently, the models can become outdated and in most cases loose their value – models do not correspond to the final implementation, possible changes are more and more demanding and it may consequent less productivity in the complex systems design. We base our approach to the system development on simulation models which have a proper formal background and can be integrated into target application with no need to generate a code. Thus, we start with simulation models but during the development process we are obtaining more and more adequate application. The models we use are based on Object-oriented Petri nets formalism. Presence of models in final implementation opens a possibility to make maintenance and adaptation to changing requirements more productive.

Keywords: Modeling, Simulation, Object-Oriented Petri Nets, Model-Based Design

Presenting Author's Biography

Radek Kočí is an assistant professor at Brno University of Technology, Faculty of Information Technology, Czech Republic, and is concerned in the education of Software engineering, Operation Systems, and Java courses. His research interest includes modeling and simulation in the context of software engineering, especially an application of Petri nets, DEVS [1], statecharts [2], and other formalisms in the system design methodology. He also cooperates on modeling and simulation of agent and multi-agent systems using Object Oriented Petri Nets. He defended his Ph.D. thesis *Methods and tools for Implementing Open Simulation Systems* in 2004.

ISBN 978-3-901608-32-2

1 Introduction

The important property of software system design and development is a quality and a productivity of developed systems as well as development processes. There are many approaches to keep development processes more productive. Each such approach uses models as a basic means for description of system structure and system dynamism. Models, contrary of the programming environment, allow developer to better think about designed system with no needs for thinking out problems sequent on programming language specificity. Of course, models are part of methodologies in software engineering for many years – we may mention the *Yourdan method* of structured systems analysis and design developed by Edward Yourdan and his colleagues at the turn of 1970s and 1980s or *Unified Modeling Language* (UML) by OMG consortium in presents.

When we will go through the development process using the most popular modeling language UML, we can see that we use two basic sets of models – models describing static relationships between modeled entities (a typical example is a class diagram) and models describing selected dynamic relationships established in some conditions (e.g., a collaboration diagram, an object diagram, etc.) These models have a static character and their purpose is to make a conceptual design of solved problems enabling better understanding of the system design. Then the designers have to implement the resulting system according to the models in selected programming language and framework.

Testing and correctness checking are another topic of software system development. These activities are usually affiliated with a program created in some programming language. We first have to have an executable variant of models (a prototype) so as we are able to check the software correctness. Since we design a part of systems as models, then we implement it, and then we check it, it may lead to less productivity in the complex systems design – we must look for errors in the prototype implementation and then correct them in the models. After it, the created prototype often missed its value and we can do just one thing – to throw out.

However, present software systems are more sophisticated and more complex thus using static models is ever more exacting because of their extensiveness. There were developed methods with a view to eliminate described problems of system development. They have two essential attributes distinguishing these methods from the conventional approach. In the first place, the models are used not only as an abstract view on the developed system, but also as an executable prototype. Thus, we must use such models which can be simulated, e.g., *Executable UML* (ExUML) [3] which enriches the classic UML models with more precise semantics allowing for the model simulation. Secondly, models are to be transformed to other kinds of models or to a programming language. To ensure it, the metamodels and metamodeling has been introduced. Metamodels are models of modeling language and define semantics of language elements. As an illustration we may cite *Model Driven Architecture* methodology with the *Meta-Object Facility*, which is the OMG's adopted standard for metamodeling [4, 5]. Nevertheless, the model transformation entails some problems – since the transformation into programming languages is not fully automated, the generated code is never final and it has to be fine completed by hand. These changes are not carried back to models and the new generation can lose them. The methods covering presented attributes are generally called *Model Based Development* (MBD).

A lot of other models or paradigms is suitable for model-based design, e.g. statecharts, DEVS (Discrete Event Systems Specification), Petri Nets, or special tools (e.g. the MetaEdit system [6]). We are interested in the Object Oriented Petri Nets (OOPN) formalism and the associated PNtalk system [7, 8, 9]. PNtalk is a long-term project started in 1993 as an original attempt to bring high-level Petri nets closer to programming languages. Main goal of this experiment was to prove that formal models such as Petri nets can be used similarly to traditional programming languages during systems development. The rigorous mathematical nature of OOPN offers a potential to solve analysis and verification problems. Several experimental implementations and verification methods were developed during recent years [10].

The goal of this paper is to outline a system development approach which use models which are a bit different from the approaches based on UML-like models. The key idea is to use models not only as diagrams enabling better quality of thinking about developed system but, in the first place, as a part living through all the development stages. More plainly, the models can serve not only as a documentation but also as a workable prototype including a possibility to use them as a part of resulting application. The presented approach joins together phases of design, testing, and implementation. We obtain a method having several benefits. The correctness of designed application is tested by simulation of models with no need for code generation. The OOPN formalism has a formal background, so it is possible to check models in the way of formal verifications too. Finally, a possibility to leave models in the target application allows for debugging the application on the model basis – the application is always seen as a set of models.

2 Object Oriented Petri Nets

Several attempts to combine Petri nets and objects has been done in the nineteens, for instance Object Petri Nets [11], Cooperative Nets [12], Nets-in-nets formalism [13]. They are supported by specialized tools like, e.g., Renew [14]. Object Oriented Petri Nets (OOPN) [7], developed by our research team, is a formalism covering advantages of Petri nets and object orientation. Petri nets allow to describe properties of the modeled system in a proper formal way and the object-orientation brings structuring and a possibility of net instantiation. OOPN along with interactive incremental development allows to design systems at different levels

ISBN 978-3-901608-32-2

of abstraction using sequential reinforcements and improvements. OOPN can be interoperable with another kind of objects so that the ability to deploy models on the target application platform or to test models in a real software environment [15] gets better. We are also developing techniques of formal verification of OOPN [10].

The formalism of OOPN is closely associated with Smalltalk environment – Smalltalk is its inscription language (actions and guards are described using Smalltalk), and, moreover, the tool based on OOPN called PNtalk is implemented in Smalltalk. It implies that there can be native cooperation between OOPN and Smalltalk objects. So far, we have implemented that kind of interoperability, so that it is possible to transparently access OOPN objects from Smalltalk and vice versa.

2.1 Introduction to OOPN

Models in OOPN are organized into classes. A *class* is specified by an object net, a set of method nets, a set of synchronous ports, and a set of message selectors corresponding to its method nets and ports. Object nets describe possible autonomous activities of objects, method nets describe reactions of objects to messages sent to them from the outside, and synchronous ports allow for remotely testing states of objects and changing them in an atomic way. Classes can be specified incrementally using *inheritance*. The inherited methods and synchronous ports can be redefined and new methods and synchronous ports can be added. The same mechanism applies for object net places and transitions.

An example demonstrating the notation of OOPN is shown in Figure 1. It consists of classes C0 and C1. Class C0 contains only the object net. Class C1 contains its object net (place p and transition t), synchronous port *state:*, and methods *wait:* and *reset*.

2.2 Object net

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).

2.3 Method net

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. Method nets are *dynamically instantiated* by message passing specified by *transition actions* (Figure 2).

2.4 Synchronous port

Synchronous ports are intended for synchronous interaction of objects. The synchronous interactions (invocation of synchronous ports) are specified in transition guards as message sendings. A (sender) transition is firable only if the receiver of the message in its guard agree with it (the involved synchronous port is firable).

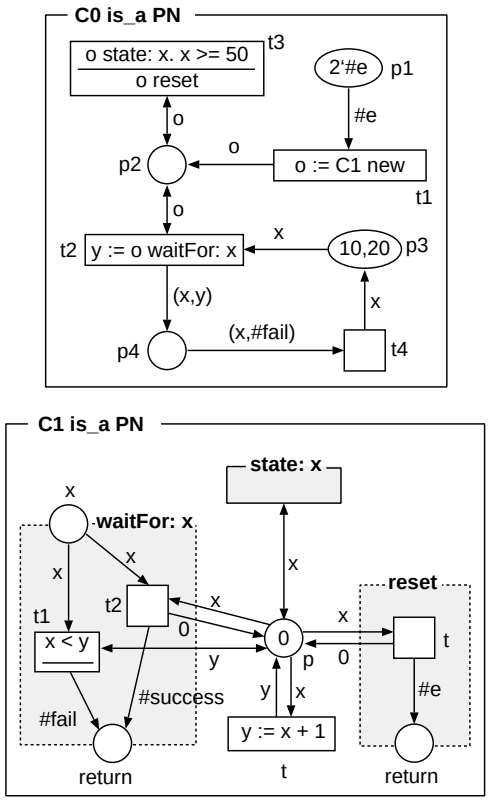

Fig. 1 An OOPN example.

The synchronous port is then *executed simultaneously* with the sender transition. The semantics of the synchronous interaction can be described as a transition which is a fusion of the sender transition and the synchronous port (respecting polymorphism and parameters binding) (Figure 3).

3 The PNtalk System

PNtalk is a simulation framework based on OOPN implemented in Smalltalk environment. Its architecture has been designed as open and metalevel and allows interoperability between OOPN and Smalltalk objects. The metalevel architecture distinguishes two architectural levels. *Domain model* describes developed system using appropriate domain paradigm, e.g., OOPN. *Meta model* describes the domain model in computational environment. The domain model has no direct representation in an implementation language, but it is transformed into special object called metaobjects. The PNtalk system architecture introduces a new meta level between the domain objects (i.e. OOPN classes and objects) and Smalltalk. This approach allows us to have full control over the domain object's structure and behavior.

We can take a look at the metalevel in two views. Firstly, we take a look at the representation of OOPN classes and objects. The OOPN classes consists of compiled nets containing compiled places and transi-

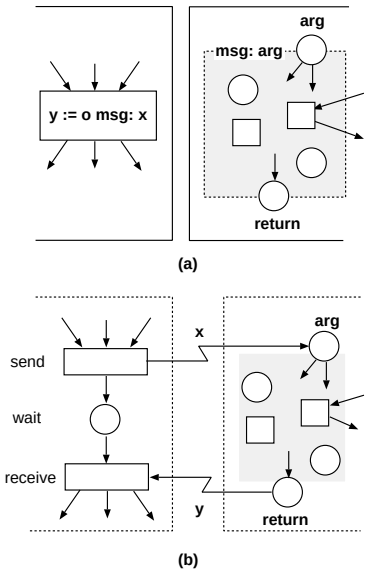

Fig. 2 Client-server interaction – syntax (a) and semantics (b).

tions. Each such element has its own metaobject describing its behavior and its state. When a new instance of OOPN class is being created then the new metaobject is established as an instance of Smalltalk class *PNObject.* Similarly to the metaobject of the OOPN class, the metaobject *PNObject* consists of *processes*. When some method is being invocated then a copy of the appropriate *compiled net* is created and executed as a part of the object's process.

The second view is the system dynamism. The metaobject *PNObject* offers a metaprotocol for controlling the simulation. The simulation, however, consists of more than one object and all these objects must share the same space. It means that there has to be a common metaobject controlling the simulation run including the time management. This metaobject is named *world* and it is implemented by Smalltalk class *PNtalkWorld.* To make simulation runing, the metaobject *PNObject* must be placed into some world – without the world, it has no dynamism.

4 The car-sharing case study

In this section, we introduce a small example which was created to illustrate a software development based on the Model Driven System Design approach (MDSD). It covers all development stages, from business process analysis, design, and model-driven code generation to the implementation of the business logic. This example has been inspired by [16].

The example describes an information system for a car-sharing company. The application and its complex development process is shown in [16]. We will concentrate on basic parts and will demonstrate a difference between using of UML-based formalisms and the OOPN formalism.

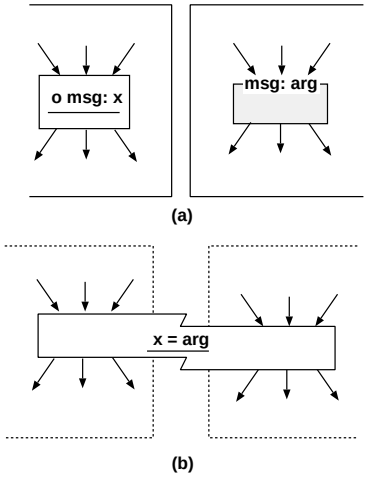

Fig. 3 Atomic synchronous interaction – syntax (a) and semantics (b).

The functionality of developed systems is usually introduced by means of *use case diagrams*. The use case diagram depicting a basic functionality of our example is shown in Figure 4. We can see that the application allows user to create a new reservation, to cancel reservation, and to edit reservation. Each of that functions requires an identification of the user (member). The new reservation and the reservation changing require further operations.

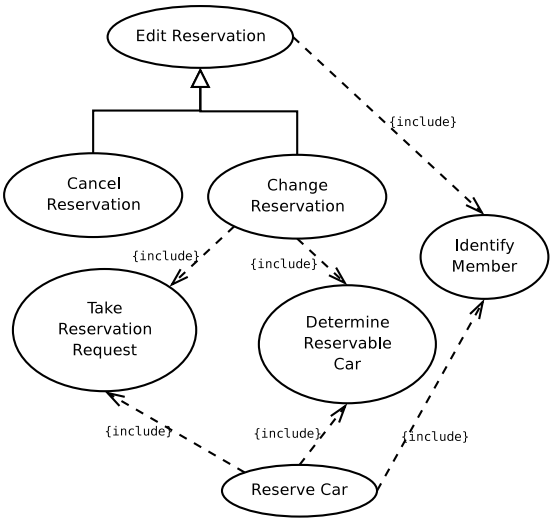

Fig. 4 Use case diagram of the car-sharing application.

We will not describe the entire design of presented example with MDSD approach. What is interesting for us now is an activity order of the application. This can be described by activity diagram in Figure 5. This diagram will serve as a demonstration of different approaches using UML-like diagrams and Object Oriented Petri Nets.

In [16], the architecture of the car-sharing application is

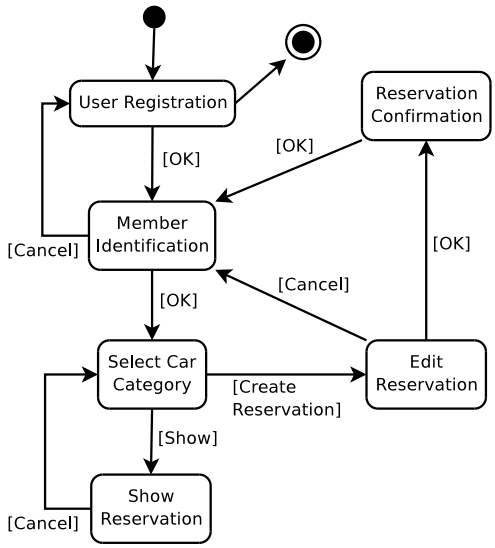

Fig. 5 Activity diagram of the navigation order in the application.

designed as a classic three-tier one consisting of a presentation layer, a process layer, and a persistence layer and it is based on J2EE framework. To apply principles of Model Driven Software Development in practise, we have to have a specialized tool which must to be able to work with extended UML including modeling constraints. The constraints are needed for the code generator. The tool has to also be conformed to used target platform (e.g., J2EE). So the models are designed with respect to the code generation in chosen platforms. Moreover, the generated code is never final and it has to be fine completed by hand. These changes are not carried back to models and the new generation can lose them.

In next chapters, we demonstrate an application of our kind of design methodology using OOPN which can eliminate the problems described above.

5 System Design Using OOPN

Formalisms like Object Oriented Petri Nets have a pure semantics and it is not necessary to enrich them by additional properties to obtain unambiguous expressions (as an example we may note the UML models and Object Constrain Language). Moreover, these formalisms can be directly interpreted and, consequently, integrated into target applications. It implies that there is no need for code generation and it is possible to debug and to really develop applications using models.

The system design based of OOPN formalism can be characterized in following points.

1. The process starts with simple models of the activity order, which is similar to workflow modeling.

2. Then we identify subnets in the models and classify them into classes.

3. The designer models various layers (aspects) of the system, such as activity models, interface models, models of shared resources, etc.

4. It is possible to switch between different views over nets, e.g., the base structure and behavior, the subnet of accessors, timeouts, etc. In fact, all these subnets form one net, but we can make it more transparent using a concept of views.

In fact, these points do not describe a sequence of activities, but represent activities done during the system design simultaneously. The system is developed incrementally, in each step we model the system activities, make decision what part is to be modeled at what layer and, if necessary, change a structure of subnets.

5.1 Model of Activity

This chapter concentrates on basic OOPN modeling. Firstly, we analyze a behavior of the registration process (see starting activities in Figure 5). This process is modeled as a net which is instantiated whenever the new user connects to the application (see Figure 6). The sequence of activity is *login*, *verify user*, and *logout*. If the user verification is successful, the net describing the user behavior is created and placed into a place *user*. If the user verification failed, the net state is moved back into the start marking so that the user can try to login again.

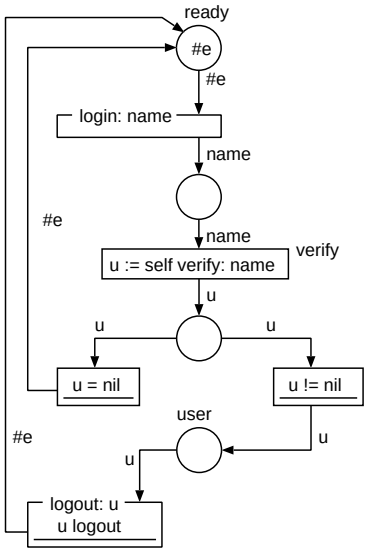

Fig. 6 The registration activity net.

The activity is modeled as a sequence of transitions or synchronous ports. While the transition firing is conditioned only by its input places, the synchronous port has to be called out in addition, analogous to the methods. The transition models *an internal event* and the synchronous port models *an event synchronized with some external event*. As an example we may take an event *login:*, which is modeled by means of synchronous port. To execute this event, the synchronous port has to be called from the net's surroundings, how we will see

in the next paragraph. The event *verify* is modeled by means of transition because it is an internal activity of the net. Of course, it can call some other methods or synchronous ports, but its execution is not conditioned by an external initiative.

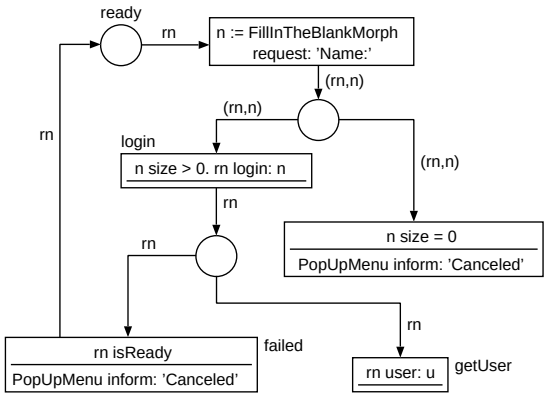

Fig. 7 The user interface net.

The user interface can be modeled as the net shown in Figure 7. When the user connects an application, the new registration net is created and placed into the place *ready*. The interface for typing user name is simulated by Smalltalk class *FillInTheBlankMorph*. If a size of the returned string *n* is greater that 0, the activities linked with registration are started. We can see two commands in the guard of the transition *login* – testing if *n size ¿ 0* and calling synchronous port *rn login: n*. Synchronous ports allow for connection of different nets in synchronous way, so that the transition *login* will be fired with the synchronous port *login:* of the net in Figure 6 at the same time.

Now the user verification is being processed (see calling the method *verify: name* in the transition *verify* in Figure 6) and the user interface net (Figure 7) is waiting for the result. If the registration process failed, the registration net is in the state represented by an anonymous token (*#e*) in the place *ready* of the net in Figure 6. If the registration process is successful, the new net representing a user behavior is created as a result of the method *verify:* and placed into the place *user*. These two different states are verifiable by means of synchronous ports *isReady* and *user:* (see calling them from guards of transitions *failed* and *getUser* in Figure 7). Nevertheless, we did not design these ports in the registration net because they are not essential for the basic workflow description. So, these ports are added into the net in the second view enabling transparent communication between nets. This second view is shown in Figure 8.

5.2 Modelled classes

So far, we have got two nets, one of them in two views. As an integral part we used synchronous ports. We have modeled registration activities as two layers – first one is a net describing a workflow of registration and second one is a net describing a model of the user interface. These layers can be encapsulated to objects described

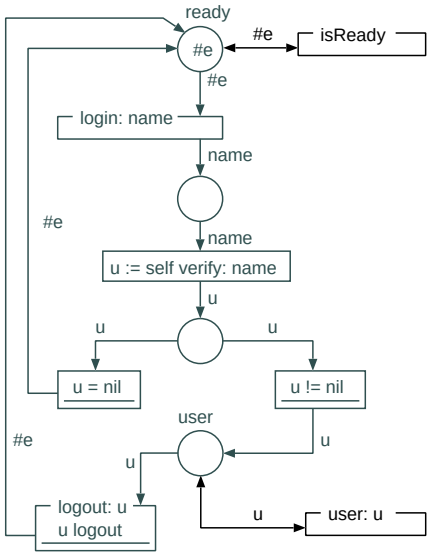

Fig. 8 The registration activity net – another view.

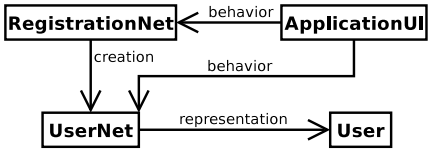

Fig. 9 Class diagram of the car-sharing application.

by classes. On the basis of acquired experiences during the system modeling, we can identify other classes and their associations, as it is shown in Figure 9:

- *RegistrationNet* and *UserNet* describing workflows of registration and user activities.
- *ApplicationUI* modeling a user interface to an application. This part of the model can be replaced by real UI on more complex model of UI in later development stages.
- *User* representing one user and keeping information about him. This class can be described either in OOPN formalism or in Smalltalk. Objects of the class *User* are stored in the persistence layer which can be either modeled by OOPN formalism or some database can be used.

All the presented nets are modeled as object nets of appropriate classes. Thus, the nets presented in Figure 6 and Figure 8 form the object net of class *RegistrationNet* and the net presented in Figure 7 is a part of the object net of class *ApplicationUI*.

5.3 Model of User Activities

An activation of synchronous port *user: u* has a side effect – if the variable *u* is free, then the object placed in the place *user* is bound to the variable *u*. Thus, it checks if the registration process is successful and also gets an

object describing the user behavior. This object is an instance of class *UserNet* and its object net is shown in Figure 10.

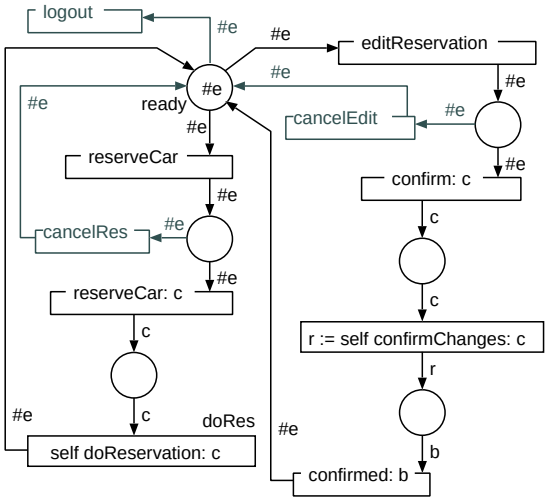

Fig. 10 Object net of the class *UserNet*.

We will continue with the user interface workflow described in Figure 7. For this case, we use a bit different approach. The interface is implemented in Smalltalk as class *UserForm* and it is accessed from the user interface net as described in Figure 11.

When the transition gets a user net *u*, it calls a message *openFor:user:* of the class *UserForm* and passes references to the user interface net (*self*) and to the user net (*u*). The message creates a new object ensuring a graphical user interface for the car reservation listing, editing, and creation of new ones.

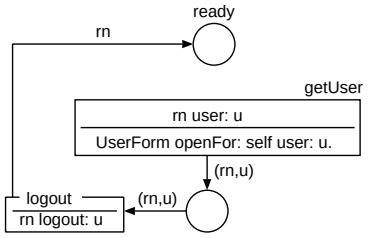

Fig. 11 The user interface net – a continuation.

Once instantiated, the object shows a list of reservations for the user *u*. The user (a person who works with the user interface) has three options – to reserve a new car, to edit a reservation, or to logout. Let us investigate the first option which can be initiated, for instance, by pressing button which invokes the method *reserveCar* of the class *UserForm* (see code in Figure 12). As a first operation, the method calls the synchronous port *reserveCar* of the user net *u*. To execute it, the method *processPort:* from metaobject protocol is used (if the OOPN reference is passed outside of the PNtalk environment, the metaobject is accessible via this reference). Now, two situations can happen. The user confirms new reservation and the method *reserveCar:* of the class *UserForm* is called. It calls synchronous port *reserveCar:* with an argument of the reserved car object *c*. It causes that the car is marked as reserved and the user net is moved to a state *ready* (see the transition *doRes* in Figure 10). The method *reserveCar* opens a list of reservations as a last operation. If the user cancels the reservation process, the method *cancelCarReservation* is called. It calls synchronous port *cancelRes* (the user net is moved to a state *ready*) and opens a list of reservations. Now we are in the start state and the user has three options again.

```
reserveCar
  u processPort: #reserveCar.
  self openEditor.

reserveCar: c
  u processPort: #reserveCar:
    values: {c}.
  self openList.

cancelCarReservation
  u processPort: #cancelRes.
  self openList.
```

Fig. 12 Selected methods of the class *UserForm*.

5.4 Model of persistence

We supposed that users and reservation records are somewhere in the storage so far. Now, we outline a possibilities how to model a persistence layer. The method *verify:* of the class *RegistrationNet* (see Figure 6) is taken as a demo example. Basically, we have three ways to model a storage of users and an access to it.

- Using the pure OOPN formalism and its expression potential.
- Using a combination of the OOPN formalism and Smalltalk language.
- Using a combination of the OOPN formalism and Smalltalk language to access some database system.

The first way is shown in Figure 13 (at the top) and the second method is shown in Figure 13 (at the bottom). Both cases use the place *storage* of object net to store information about users. The first method uses this place to store pairs *(name, user)* and is looking for users in accordance to its names by means of the OOPN binding mechanism. The method has two transitions, one for successful selecting (*t1*) and one for unsuccessful selecting (*t2*). If there is a pair whose *name* matches the passed *name*, the transition *t1* is firable and the found object is assigned to a variable *user*. If there is no such pair, the transition *t2* fires and puts a value *nil* to the return place. Note, that the special inhibitor arc is used between the place *storage* and the transition *t2*.

ISBN 978-3-901608-32-2

The second way uses the place *storage* to store an object of Smalltalk class *Dictionary*. Transitions *t1* and *t2* then use ordinary methods (*c containsKey: name*) in their guards to make out that the user exists or doesn't. If it does, the transition *t1* fires, gets appropriate object from the dictionary, creates an object as an instance of class *UserNet* and puts this object into the return place. If it doesn't, the transition *t2* fires and puts a value *nil* to the return place.

The third way (not shown) is similar to the second one in principle. The storage is not represented as an instance of Smalltalk class *Dictionary*, but it is an object ensuring an access to selected database system.

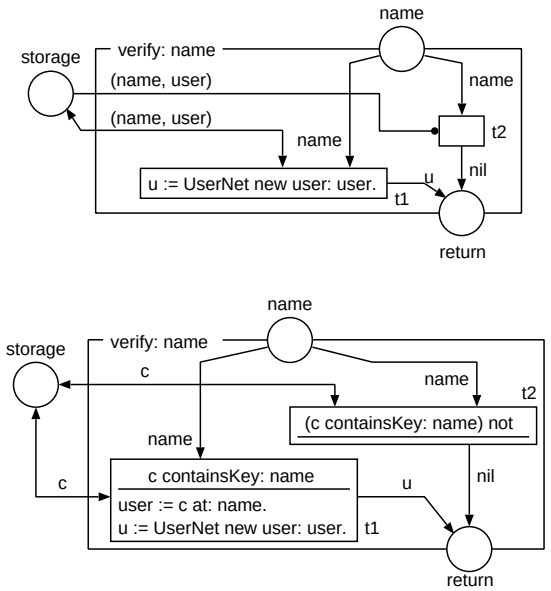

Fig. 13 The method net *verify:*.

5.5 System Simulation

System simulation is an integral part of our methodology. Contrary to the UML models, the developer works with simulation models at any time. Simulation helps to find errors in the design immediately as well as to analyze a behavior of the designed system in specific situations (e.g., traffic simulation, throughput of the system working with processes, etc.) The simulation can be interactive (it means that the developer makes decision about what transition is to fire, what value is to be bound etc.) or automatical, e.g., automatical testing.

6 Conclusion

The system design based on models with a subsequent code generation helps developers to improve the developed process efficiency, but there is still a drawback which stem from the separation of models (design phases) and the application (the models implementation).

The PNtalk system has been designed as a tool suitable for system development based on modeling and simulation. Their characteristic is to use formal models (OOPN) in conjunction with incremental and iterative development process. It allows for high-quality and rapid development and for combination of modeled and real components. For instance, suppose that we are developing a system for robotics control. Then we may first model the actuator (and we are able to verify the correctness of our algorithms and concepts by simulation) and then we can replace this model by e.g. the real robotic arm (whereas, of course, the next testing follows). The another example is a system driven by a workflow with a WWW interface. We are able to simulate this interface including its inputs and responses so that we may concentrate on the system logic. During the development we can concurrently design the WWW interface and finally we replace the simulated interface by its developed real variant.

The important idea is also *model continuity*. It makes the tendency towards an elimination of generating the source code from models. The system is developed incrementally, in each step the models are being improved and combined with real components. Finally, the key part, in particular the control of the system logic, is kept in the system as its integral part. For example, we are able to develop the car-sharing application in the OOPN formalism, the user interface in Smalltalk (using, e.g., *SeaSide framework*), and to get functional application by their integration.

In the presented approach we have used several layers – business logic (workflow), presentation, and persistence. The models of the layers are synchronized by means of synchronous ports. Connection to the surroundings of the model is accomplished by means of inherent ability of PNtalk to interoperate with Smalltalk. This way the user interface and databases can be accessed.

The presented experimental methodology and tool enrich the system development process with a number of elements which can improve a quality and productivity of it. The developer programs by modeling, there is no code generation because the model is a code. There is no difference between programming environment, modeling environment, or testing environment – every development tasks are cover by one environment.

For the present, this methodology including the tool serves for experiments and demonstration of the idea of our approach to the system development processes. It is also not integrated with industrially used technologies and standards which can be seen as a drawback of it. The advancement of the methodology and tools, and the approximation to standard applications are topics of further research.

7 Acknowledgement

This work was supported by the Czech Grant Agency under the contracts GA102/07/0322, GP102/07/P306, and Ministry of Education, Youth and Sports under the contract MSM 0021630528.

8 References

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

[2] D. Harel and M. Politi. *Modeling reactive systems with Statecharts: The Statement Approach*. McGraw-Hill Companies, 1998.

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

[4] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture – Practice and Promise*. 1st edition. Addison-Wesley Professional, 2003.

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

[6] MetaCase. Domain-Specific Modeling with MetaEdit+. http://www.metacase.com, 2007.

[7] M. Češka, V. Janoušek, and T. Vojnar. PNtalk - A Computerized Tool for Object Oriented Petri Nets Modelling. In *Proceedings of the 5th International Conference on Computer Aided Systems Theory and Technology – EUROCAST'97*, pages 229–231. Las Palmas de Gran Canaria, ES, 1997.

[8] V. Janoušek. *Modelování objektů Petriho sítěmi*. PhD thesis, FEI VUT, Brno, CZ, 1998.

[9] V. Janoušek and R. Kočí. PNtalk Project: Current Research Direction. In *Simulation Almanac 2005*. Faculty of Electrical Engineering, Praha, CZ, 2005.

[10] M. Češka, V. Janoušek, R. Kočí, B. Křena, and T. Vojnar. PNtalk: State of the Art. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components, and Agents*, pages 301–307. Hamburg, DE, 2006.

[11] C. A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *Proceedings of the Application and Theory of Petri Nets*, volume 935. Spinger-Verlag, 1995.

[12] C. Sibertin-Blanc. Cooperative Nets. In *Proceedings of Application and Theory of Petri Nets*, volume 815. Springer-Verlag, 1994.

[13] 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, June 2005. Proceedings*, pages 148–167, 2005.

[14] O. Kummer, F. Wienberg, and et al. An extensible editor and simulation engine for Petri nets: Renew. In *Applications and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings*, volume 3099, pages 484–493. Springer, jun 2004.

[15] V. Janoušek and R. Kočí. Towards Model-Based Design with PNtalk. In *Proceedings of the International Workshop MOSMIC'2005*. Faculty of management science and Informatics of Zilina University, SK, 2005.

[16] T. Stahl and M. Volter. *Model-Driven Software Development*. John Wiley & Sons Ltd., England, 2006.

ISBN 978-3-901608-32-2