System Design with Object Oriented Petri Nets Formalism

Radek Kočí and Vladimír Janoušek
Brno University of Technology
Faculty of Information Technology
Bozetechova 2, 616 00 Brno, Czech Republic
{koci,janousek}@fit.vutbr.cz

Abstract

The actual trend in the research of system design aims at an efficiency and safety of developing processes as well as at the quality of resulted systems. There were investigated and developed many methodologies of system design based on models – they are known as Model-Based Design. These methodologies use executable semi-formal models allowing for transformations including code generation in selected language. Nevertheless, the further development or debugging by means of prime models is impossible. This paper brings an outline of our approach to Model-Based Design based on the Object Oriented Petri Nets formalism allowing for clear modeling, the possibility to check correctness by simulation techniques as well as by formal verifications. The model is an executable program valid through all development stages including the target application. The paper depicts the basis of the formalism and used techniques.

1. Introduction

The actual trend in the research of system design aims at an efficiency and safety of developing processes as well as at the quality of resulted systems. The processes usually use models as a basic means for description of systems. 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. These models can be classified into two sets—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.) Nevertheless, 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. When we wish to check dynamic properties of developed system or to get the target application, we have to implement an executable prototype or system according to the models in selected programming language and framework.

The next question is about used methodologies. The classic approach to system design is based on development phases which are performed in sequence: analysis, design, implementation, and testing. This methodology is simple for its realization, but is not sufficient for design of complex systems, because the system requirements change during the development process and the classic methodology are not very conform with it. As an answer to this the incremental and iterative design methodologies were developed, e.g. Unified Process or Agile Methodologies. They allow for concurrent execution of design phases, the changes are done by small steps, lays stress on testing, etc. These methodologies are more successful for ordinary applications, but have their limitations. The main problem is that the implementation and testing are separated from the design phase.

That is why the new methodologies and approaches are investigated and developed for many years. They are commonly known as *Model-Driven Software Development* or *Model-Based Design* (MBD) [6]. An important feature of these methods is the fact that they use executable models. The designer creates models and checks their correctness by simulation, so that there is no need to make a prototype. The most popular methodology is *Object Management Group's Model Driven Architecture* (MDA) based on Executable UML [10]. These methodologies use semi-formal models allowing for model transformations including code generation. Since the code has to be finalized manually with no backward transformation, it entails a possibility of semantic mistakes or imprecision between models and transformed code. Moreover, the further development, debugging and investigating of target application by means of prime models is impossible, so the sig-

978-0-7695-3372-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ICSEA.2008.33

nificance of model has declined.

Our research group is interesting in modeling and simulation techniques applied on the system design [4, 5]. We use formal models as an executable program valid through all development stages including the target application. In comparison with semi-formal models, formal models bear the clear and understandable modeling, the possibility to check correctness not only by simulation techniques, but also by means of formal verifications [2]. A lot of model paradigms is suitable for Model-Based Design, e.g., Statecharts, DEVS, Petri Nets, or special tools, e.g., the MetaEdit system [9]. We have developed a tool named PNtalk which is based on the formalism of Object Oriented Petri Nets [3]. This formalism and tool support theoretic and experiment research in the presented approach of the system design.

The goal of this paper is to outline the system development approach based on formal models, namely OOPN formalism, and simulation techniques. The key idea is to use models as a part living through all the development stages. The correctness of designed application is tested by simulation of models with no need for code generation. We will show these techniques in an example of the conference system case study.

2. Object Oriented Petri Nets

Several attempts to combine Petri Nets and objects has been done in the nineteens, for instance Object Petri Nets [8], Cooperative Nets [11], Nets-in-nets formalism [1]. They are supported by specialized tools like, e.g., Renew [7]. Object Oriented Petri Nets (OOPN) [3], 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.

The formalism of OOPN consists of Petri Nets organized in classes. Every class consists of an object net and a set of dynamically instantiated method nets. Object nets as well as method nets can be inherited. Inherited transitions and places of object nets (identified by their names) can be redefined and new places and/or transitions can be added in subclasses. Inherited methods can be redefined and new methods can be added in subclasses. A token in OOPN represents either a trivial object (e.g., a number or a string) or an instance of some Petri Net.[1] A *class* is specified by an object net, a set of method nets, a set of synchronous ports, a set of negative predicates, and a set of message selectors corresponding to its method nets and ports.

[1] As defined in the original formalism. Currently, the formalism is extended and it is possible to use another kind of objects as tokens.

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 each of them has a special 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 1).

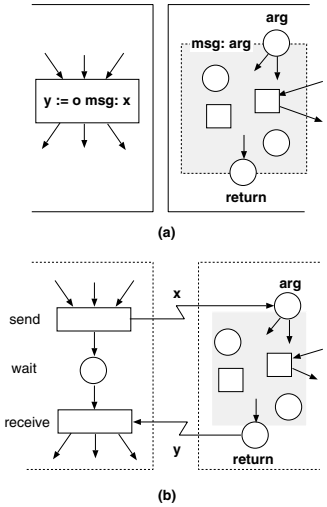

Figure 1. Client-server interaction – syntax (a) and semantics (b).

Synchronous ports are intended for synchronous interaction of objects. The synchronous port is a hybrid of method and transition that allow for synchronous interactions of objects. In order to be fired, the synchronous port has to be called (a method concept) and has to be firable (a transition concept). Synchronous port can be called only from a transition guard. The transition can be fired only if all called synchronous ports are able to fire. The synchronous port can be called with bound or unbound variables. In the case of calling synchronous ports with unbound arguments, the potential bindings of synchronous port are resolved and, if such binding exists, the arguments are bound to the resolved values. A special variant of synchronous port is *negative predicate*. Its semantics is inverted—calling transition is firable if the negative predicate is not firable.

An example illustrating the important elements of the OOPN formalism is shown in Figure 2. 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 the method *doFor:*. The semantics of the method *doFor:* execution is to randomly generate *x* numbers and return their sum.

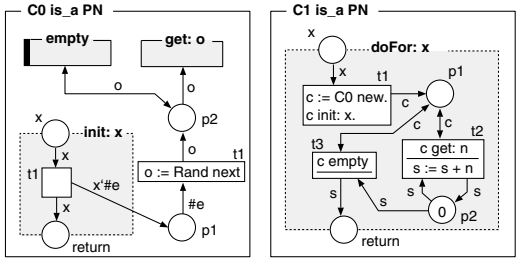

Figure 2. An OOPN example.

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 modeling and simulation framework based on OOPN named 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. So that it is possible to get Smalltalk object as the token in OOPN. More detailed description of the PNtalk framework can be found in [3].

3. Model Based Design with OOPN

We will demonstrate the system design concepts on the case study of the conference system. To follow a goal of the paper, we will not deal with whole case study, we only select certain parts of designed models to show modeling and simulation techniques applied in the system design. The whole model will be published elsewhere.

3.1. The Design Process

The design process starts with analysis of the target domain. We will distinguish persistent entities and the behaviors. Some of these entities are passive objects (e.g., *the paper* and *the review*), other ones are active subjects (e.g., *the author* or *the reviewer*). The design can be characterized in following points.

- The identification of persistent objects and their classification into two groups—passive objects and active subjects.
- The identification of roles of each active subject and classification of subjects into classes. In our example, the author or the reviewer are active subjects having the same basis—both are persons playing different roles in the system. Moreover, one person can play a role of author and reviewer together. So we can classify them into the class `Member` and identify two roles: *author* and *reviewer*.
- The classification of passive objects into classes. In our example, the paper or the review are passive objects. These kind of objects has its state and offers only a protocol to get this state or to modify this state. It has no own activity, i.e., it has no active role in the system. The passive objects can be modeled by used formalism of OOPN or by means of other language (e.g., Smalltalk).
- The classification of roles into classes. Each role defines a set of behaviors, we can identify the main task supported by several sub-tasks. The main task is modeled by object net, the sub-tasks by method nets.

3.2. The Model of Behavior

The behavior of every objects, subjects, and roles is defined in two ways which are combined. Firstly, the workflow modeled as a sequence of fired transitions or synchronous ports. Secondly, the state detection modeled by synchronous ports and negative predicates. We will demonstrate these ways in different nets.

In our example, the allowed actions depend on the state of the conference system. So we have to have another one subject—the conference itself. It stores an information about a state of the conference and defines operations allowing to change this state. For simplicity we suppose following states: *open*, *review*, *final*, and *close*.

The conference object stores registered persistent objects (members) too. The object net `Conference`[2] is shown in Figure 3. Members are identified by their name with no authentication in our example. The object net contains means for state testing (synchronous port `phase:`) or getting stored member with passed name (synchronous port `member:named:`). The negative predicate `notMemberNamed:` serves for execution of variants of wrong member identification (see method net `verify:as:` in chapter 3.3).

Each member must log-in to the system before any action. The process of member identification (we leave out the registration) is modeled as a net which is instantiated

[2]Since each OOPN class consists of just one object net, this net identify the class at the same time, and it is also possible to identify this object net by the class name.

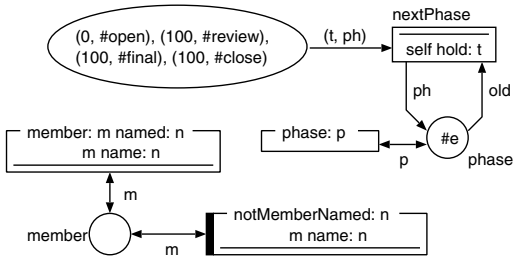

Figure 3. The conference object net.

whenever the new user connects to the application (see Figure 4). 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 *verifiedUser*. If the user verification failed, the net state is moved back into the start marking so that the user can try to log-in again.

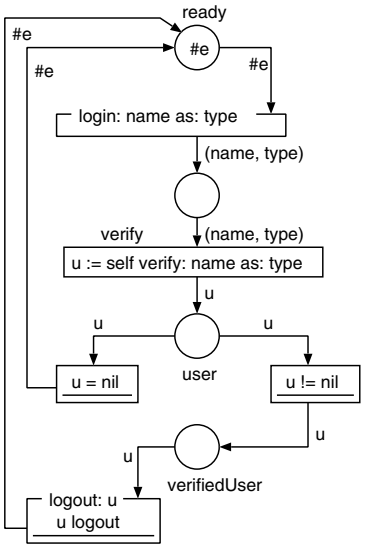

Figure 4. The log-in activity net.

3.3. Internal and External Events

The activity (workflow) 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 as well, 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:as:*, which is modeled by means of synchronous port. To execute this event, the synchronous port has to be called from the net's surroundings, e.g., another net or Smalltalk code (see chapter 3.6).

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. The verification process is modeled by the method net `verify:as:` (Figure 5). If the user verification successes (i.e., there is a member with passed name which can play the author role), this net returns a net corresponding to the class describing a behavior of the author role (the class `AuthorNet`). Otherwise it returns `nil`.

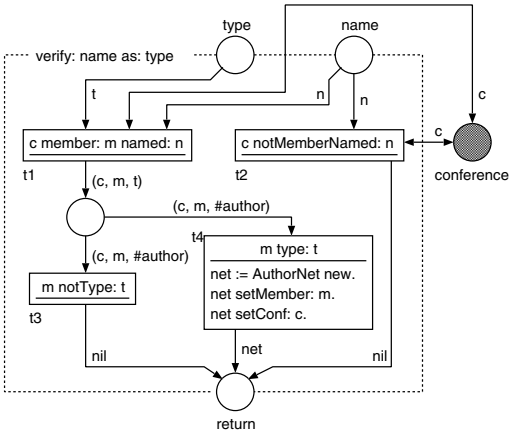

Figure 5. The method net verify:as:.

3.4. The Model of Role Behaviors

As we have seen in previous chapter, the specific role of some subject is created as an instance of appropriate class. The newly created object wraps the subject for which this role is specified, so that the role object can communicate to wrapped subject. The character of presented case study points out the modeling way for object nets of most of the roles. It is mainly the state detection, as we can see in Figure 6. The object net `AuthorNet` consists of two places, `member` containing an object of author, and `conference` containing an object of conference. It defines following synchronous ports and negative predicates. The synchronous port `member:` tests if this net represents a role of the passed member object or allowing to get the member object. The synchronous port `paper:` tests if the author net contains passed paper or allowing to get the paper (perhaps even papers, if the author submitted more papers)—because these objects are stored in the wrapped persistent object, this operation is delegated to this object. The synchronous port `canPutPaper` tests if it is possible to put paper to the system. The negative predicate `cannotPutPaper` tests if it is not possible to put paper to the system.

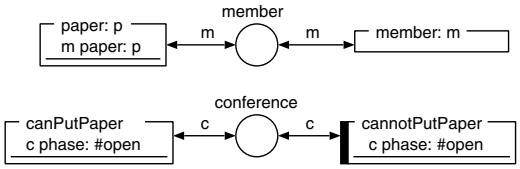

Figure 6. The author net.

We also have to define ways for the state modification. In principle, there are two ways how to do it—to use synchronous ports or method nets. Both of them have its advantages and disadvantages. In general, to set a state or to put some object the method net seems to be better. The class `AuthorNet` defines method nets `setConf:`, `setName:`, and `addPaper:` (this method serves for paper insertion and is shown in Figure 7).

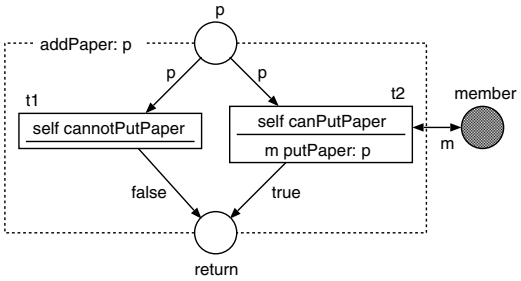

Figure 7. The addPaper: method net.

3.5. Testing

The important part of system design is checking of correctness. The most common way is testing, preferably automated testing. If we have the system described in formalisms such as OOPN, we can prepare scenarios of behavior as a workflow of transitions and synchronous ports including a check on current state. Let us have a testing net shown in Figure 8. This example tests following actions: to create a conference object, to create several (three, in the concrete example) member objects, if there is a member named *2* then the author net is created and if the conference state allows to put a paper, we put it. The first scenario assumes that the conference is in the state *open*. Then the scenario has to look like following example:

$$\{p1(3)\}t1\{p1()\}\{p2((@Conference, 3))\}$$
$$t2\{\}t2\{\}t2\{\}t3\{\}t4\{\}t6\{\}t7\{p6(\#p1)\}$$

This scenario says that, at the start, the place `p1` contains a number *3*. Then the transition `t1` is fired, the place `p1` will be empty, and the place `p2` will contain a pair of

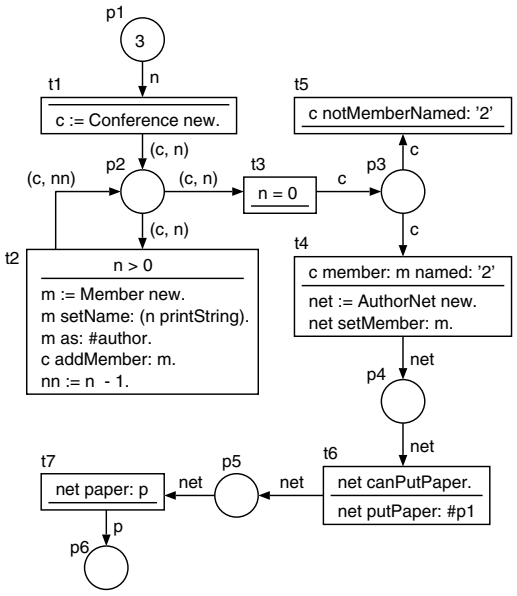

Figure 8. The test author net.

an object of the class `Conference` and a number of generated members. Then the transition `t2` will be fired for three times (the state we will not check now), etc. What is interesting and important, these scenarios can be set to the framework. The framework is able to compare this implied behavior with its simulation and quickly identify the place of different behavior or state.

Each scenario is assigned to the some net. It is also possible to include conditions to the scenario, because the current trace of run depends on the current state. This situation is shown in following scenario (it reflects the method net in Figure 7):

$$\{conference(\$c)\}$$
$$?\{\$c.phase(\#open)\} : t2\{return(true)\}$$
$$?\{not\ \$c.phase(\#open)\} : t1\{return(false)\}$$

These scenarios were presented in formalized notation, but they can be used in conjunction with graphical models in the framework, and, thus, in more comfortable way.

3.6. Model Continuity

The formalisms like OOPN 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 [4]. The important idea behind the presented approach is *model continuity*. 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 conference system in the OOPN formalism, the user interface in Smalltalk (using, e.g., *SeaSide framework*), and to get functional application by their integration.

The PNtalk architecture results from principles of meta-level and reflective architectures [3]. It introduces special objects called meta-objects allowing us to have a full control over model manipulation and run. The example of using meta-object protocol to interfacing model from Smalltalk environment is shown in Figure 9. This example shows a code placed in the method for member log-in. The object variable `conf` refers to the object of class `Conference` presented above (i.e. described by OOPN). In this example, we get a special meta-object allowing for object manipulation at the synchronous (or negative) ports level. Then we can test synchronous port by sending the message having the same name. If the testing is successful (it means the port can be fired), we can fire it (the message `perform`) and, herewith, move the net to the new state. Otherwise the log-in action failed—the member is not registered or cannot play the role of author.

```
net := RegistrationNet new.
net setConference: conf.
np := net asPort.
res := np login: name as: #author.
res
  ifTrue: [ res perform ]
  ifFalse: [ "login failed ..." ].
```

Figure 9. The model interfacing.

4. Conclusions

The paper has outlined basic principles of the Object Oriented Petri Nets formalism (OOPN) and its usability in the system design. OOPN modeling combines imperative and declarative programming techniques. The system behavior is modeled as sequences of events, the event can be conditioned by other events or declared state of the system. To each event, the sequence of commands can be assigned.

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. It makes possible to check developed systems by formal verification as well as by simulation techniques. Moreover, OOPN are 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 [4] gets better. Last but not least, the important property is a possibility to leave models in the target application which means that the application can be debugged on the model basis—the application is always seen as a set of models so that the advantages of models usage are kept during whole software life-cycle. All of this allow for safer and quicker development of software systems.

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.*

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, June 2005. Proceedings*, pages 148–167, 2005.

[2] 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.

[3] V. Janoušek and R. Kočí. Towards an Open Implementation of the PNtalk System. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation*. EUROSIM-FRANCOSIM-ARGESIM, Paris, FR, 2004.

[4] 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.

[5] V. Janoušek and R. Kočí. Simulation and design of systems with object oriented petri nets. In *Proceedings of the 6th EUROSIM Congress on Modelling and Simulation*, page 9. ARGE Simulation News, 2007.

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

[7] 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.

[8] 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.

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

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

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