OBJECT ORIENTED PETRI NETS – MODELLING TECHNIQUES CASE STUDY

R. KOČÍ, V. JANOUŠEK, and F. ZBOŘIL, jr.
Faculty of Information Technology
Brno University of Technology
Bozetechova 2, Brno, Czech Republic
Email: koci@fit.vutbr.cz

***Abstract*—The importance of models used in the area of system design is growing. There were investigated and developed many methodologies of system design based on models – they are known as Model-Based Design. These methodologies use semi-formal models and need for model transformations to check correctness or to get a final application. Formal models, by contrast, allow for clear modelling and checking 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. There are several suitable formalisms, one of them is a formalism of Petri Nets, especially its object variant, Object Oriented Petri Nets (OOPN). The paper deals with application of OOPN formalism in the area of system design and demonstrates modelling techniques of OOPN on the case study of conference system design.**

***Keywords*—Modelling, Simulation, System Design, Model Based Design**

I. INTRODUCTION

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 Modelling Language* (UML) by OMG consortium in presents. These models usually have 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.

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) [1]. 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 development methods allow for semi-automatic translation of checked models to implementation language, i.e. the code generation. The most popular methodology is *Object Management Group's Model Driven Architecture* (MDA) based on Executable UML [2]. This kind of methodologies uses semi-formal models and assumes that models will be transformed. Nevertheless, the result has to be finalised manually, so it entails a possibility of semantic mistakes or imprecision between models and transformed code. Moreover, the debugging and investigating of real application by means of prime models is impossible, so their significance has declined. In comparison with semi-formal models, formal models bring the clear and understandable modelling, the possibility to check correctness not only by simulation techniques, but also by means of formal verifications.

There are model paradigms suitable for application in the area of system design, e.g., Statecharts, DEVS, Petri Nets, or special tools, e.g., the MetaEdit system. Petri Nets have several desired properties to model and design software systems: pure description of concurrency, visual representation of the models and strong mathematical background allowing automatic checking of model properties. In the

nineteens, several variants of Petri Nets have been developed, for instance Object Petri Nets [3], Cooperative Nets [4], or Nets-in-nets formalism [5] supported by specialised tools such as, e.g., Renew [6].

The paper is aimed at modelling techniques of special variant of Petri nets—Object Oriented Petri Nets (OOPN) [7], [8]. It is a part of design methodology allowing for safer and quicker development of software systems [9], [10], [11]. OOPN modelling combines imperative and declarative programming techniques. The system behaviour is modelled 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.

There were developed the tool named PNtalk/SmallDEVS [12], [8] for checking of special approaches and principles of model-based design. The tool allows for meta-level manipulation with models as well. It can be very useful in design of self-evaluated systems as well as in testing and debugging of designed systems. The modelling techniques of OOPN will be demonstrate on the system design. We will show basic constructs and their advantage in the model design.

II. OBJECT ORIENTED PETRI NETS

Object Oriented Petri Nets (OOPN) is a formalism covering advantages of Petri nets and object orientation. Petri nets allow to describe properties of the modelled system in a proper formal way and the object-orientation brings structuring and a possibility of net instantiation.

Object Oriented Petri Nets (OOPN) consist of Petri nets organised in classes. Every class consists of an object net and a set of dynamically instantiable 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 class described by OOPN. Each such object (instance) consists of one instance of the appropriate object net and possibly several concurrently running instances of invoked method nets.

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.

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 set of parameter places and a return place. Method nets can access places of the appropriate object net in order to allow running methods to modify states of the object which they are running in. Method nets are *dynamically instantiated* by message passing specified by *transition actions* (Figure 1).

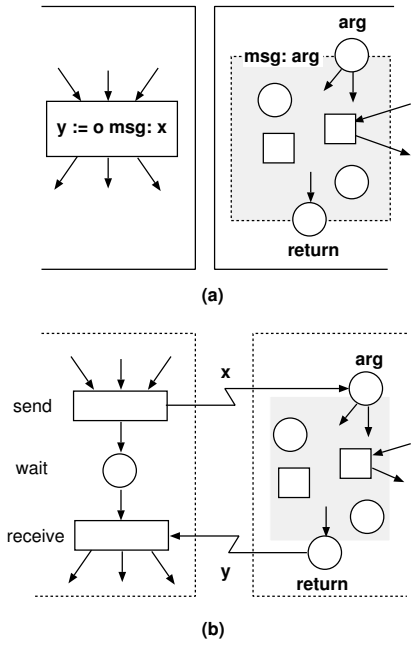

Fig. 1. Client-server interaction – syntax (a) and semantics (b).

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. It means that synchronous ports can be used for firability testing as well as for getting objects stored in places of other objects. A special variant of synchronous port is *negative predicate*. Its semantics is inverted – the calling

transition is firable if the negative predicate is not firable.

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 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.

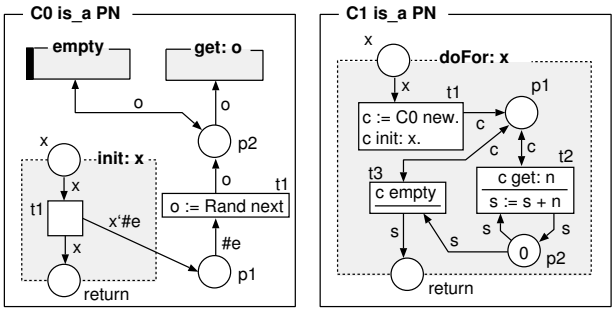

Fig. 2. An OOPN example.

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 tool based on OOPN called PNtalk [12] 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 can be found in [8], [12].

III. 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 [13], [11]. The important idea behind the presented approach is *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 conference system in the OOPN formalism, the user interface in Smalltalk (using, e.g., *SeaSide framework*), and to get functional application by their integration.

A. The PNtalk/SmallDEVS Architecture

PNtalk/SmallDEVS is a simulation framework implemented in Smalltalk environment. Its architecture has been designed as open and meta-level allowing to experiment with models at higher levels as well as the combination of more paradigms [13], [14]. At the present time we have developed the PNtalk system based on OOPN and SmallDEVS based on DEVS formalism, whereas SmallDEVS forms the basic hierarchical simulation framework to which the PNtalk system is incorporated.

The paper is interested in OOPN formalism and the meta-object facility to high-level working with models is one of the used techniques. Hence we introduce the PNtalk meta-level architecture. It distinguishes two architectural levels. *Domain-model* describes developed system using appropriate domain paradigm (OOPN in our case, but the architecture enables to extend it to another paradigms). Domain models are transformed into representation suitable for simulation – *meta-models*. The 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 meta-objects. This approach allows us to have full control over the domain object's structure and behaviour.

B. The Meta-Objects Composition

We can take a look at the meta-level of PNtalk in two views—the first one is the representation of domain concepts (classes, objects, processes, etc.). The basic meta-object composition is depicted in Figure 3. Each domain concept has its class at the meta-level (e.g., the concept of PNtalk class is represented by the class named `PNClass` at the meta-level). The concrete domain elements (e.g., PNtalk class named `C0` or the instance of the PNtalk class `C1`) is represented by instances of appropriate classes at the meta-level (i.e., instance of `PNClass` or instance of `PNObject`).

When a new instance of the PNtalk class is being created then the meta-object `PNObject` is established.

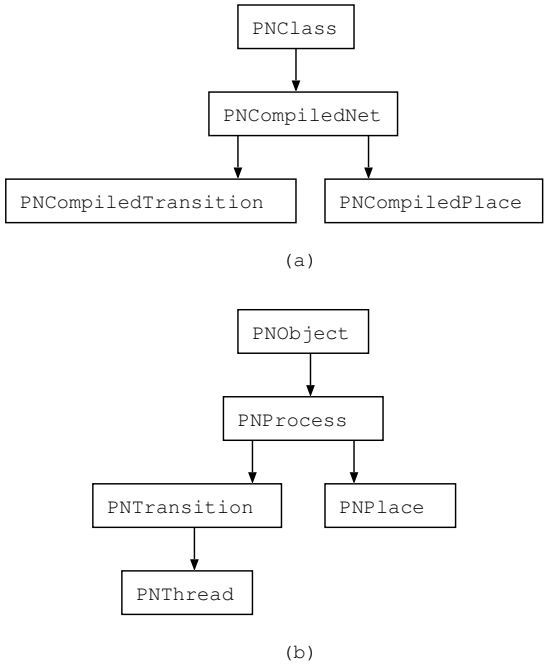

Fig. 3. The composition of the meta-objects *PNClass* and *PNObject*.

The meta-object `PNObject` consists of *processes* (see `PNtalkProcess`). When some method is being invoked then a copy of the appropriate *net* (see `PNtalkNet`) is created as the object's process. The state (marking) of place comprises object references, the state (marking) of transition comprises its invocations (i.e. fired transition). Such each invocation we called thread and it is represented by class `PNtalkThread`.

C. The System Dynamism

The second view is a system dynamism. The meta-object `PNObject` offers meta-protocol for controlling the simulation over it. 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 meta-object controlling the simulation run including the time management. This meta-object is named *world* and it is implemented by Smalltalk class `PNtalkWorld`. So as the meta-object `PNObject` to run, it must be placed into some world – without the world, it has no dynamism.

IV. META-OBJECT PROTOCOL

We will demonstrate a using of meta-object protocol on the very simple example. Let us suppose that we have a model such as presented in Figure 4. Now the question is how to check the correctness of the model semantic. There are several ways how to do it—these techniques will be demonstrate in this chapter. The other ones will be shown in the next chapters dealing with the conference system case study.

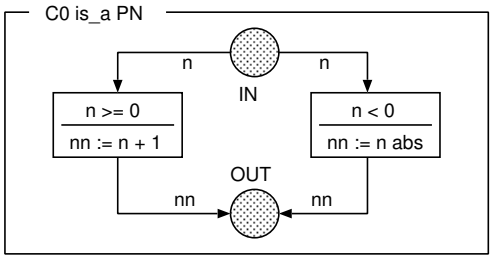

Fig. 4. The OOPN example.

A. State Modification and Introspection

The PNtalk meta-level architecture allows, among others, interoperability between OOPN and Smalltalk objects. So it is no problem to access PNtalk objects, get its meta-object representation and use meta-object protocol to affect the model. We can create an instance of class `C0` (PNtalk classes are placed in the special repository, in this example we suppose the repository is accessible via the variable `repository`), put it to the simulation world, put some object into place `IN`, do one simulation step and check the content of the place `OUT` by the code shown in the Figure 5.

```
world := PNtalkWorld new.
cls := repository componentNamed: 'C0'.
obj := cls newIn: world.
((obj meta componentID: 1)
      placeNamed: 'IN') put: 2 mult: 1.
world test. world step.
((obj meta componentID: 1)
      placeNamed: 'OUT') contains: 3.
```

Fig. 5. State modification and inspection.

B. Behaviour and Structure Modification

The external work with models can cause a problem of the model consistency (the change can cause unpredictable behaviour) in more complex models. The designer has to be more careful and the advantage of models using degrades. The other solution is to insert special behaviour into tested objects so that the class definition does not change.

Each model has both graphic and text representation. The text representation is useful for manipulation with models at the meta-level. When we got an instance, we can compile

two access synchronous ports to this instance (so that we just modify the object, not the class) and then communicate with the object via these ports, see the example in the Figure 6.

```
world := PNtalkWorld new.
cls := repository componentNamed: 'C0'.
obj := cls newIn: world.
PNtalkCompiler compile: 'sync input:
            o postcond IN(o)' to: obj.
PNtalkCompiler compile: 'sync output:
            o cond OUT(o)' to: obj.
world start.
result := obj asPort input: 2.
result ifTrue: [ result perform ].
result := obj asPort output:
                (PNVariable name: #r).
result ifTrue: [
          (result variable: #r) = 3 ].
```

Fig. 6. Structure modification.

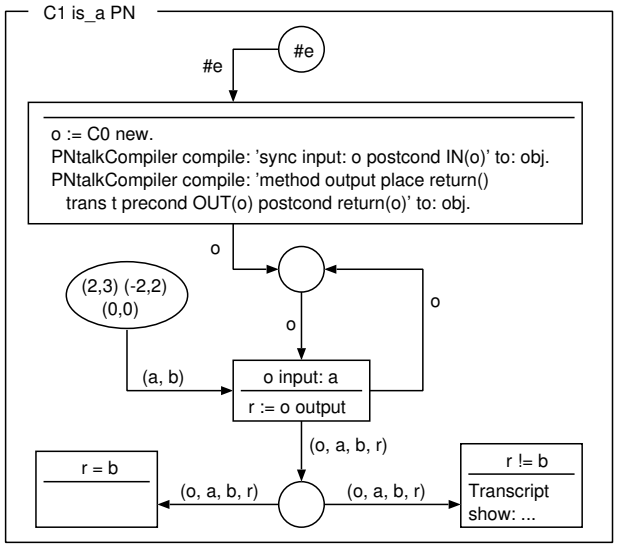

Fig. 7. Testing Net of OOPN.

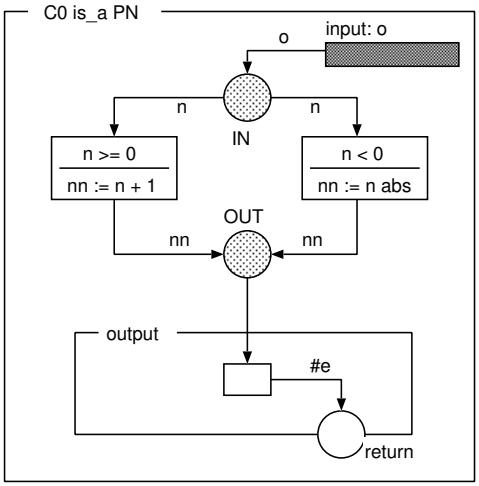

Fig. 8. Modified object representation.

C. Behaviour and Structure Modification from Models

The previous example may cause a problem of synchronisation—the question is, when is a well-timed point to check results. The solution is to compile not the synchronous port, but the method which get appropriate object. If we would use it such a way, the testing will be suspended until the method will make a result. But, if the tested nets will contain errors, the method needn't to finish, so the testing methods can lock in. This implies that it cannot be automated. Therefore, the application of method compilation directly from domain-models may be preferable in most cases, because we can use native synchronisation mechanisms of domain models. The example is shown in Figure 7. The modified object will look as depicted in Figure 8.

V. THE DESIGN PROCESS

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 modelling techniques applied in the system design.

The modelling process consists of several phases which can be processed simultaneously. The division is made because of the simplicity of developing processes description. The phases are

- *classes definition* – the identification of objects and their classification into classes,
- *behaviour definition* – the identification of the basic workflow of each object, the identification of communication channels between objects and definition support behaviour.

Different modelling techniques are related to the different phases. We will discuss it in following chapters.

VI. CLASSES DEFINITION

The design starts with analysis of the target domain. The results is a set of classes. The phase of the classes definition

is characterised by several points of activities, as described in following points.

- The object identification and their classification into two groups—passive objects and active subjects,
- the identification of roles of active subjects,
- the classification of objects and subjects into classes,
- and the classification of roles into classes.

A. Object Identification

At the first, as well as in other techniques, we have to identify objects of the problem domain, their responsibilities and relationships. The identification is based on the analysis of modelled system and we can turn some of the UML diagrams to advantage.

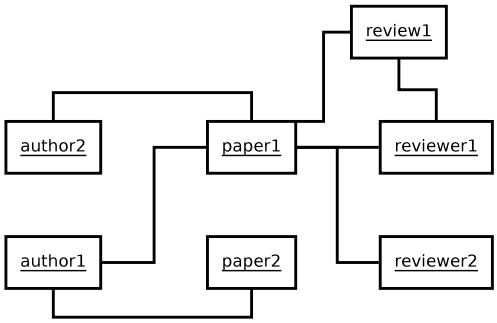

Fig. 9. The object diagram of modelled system.

To objects identification we can use the object diagram (see the Figure 9). This diagram depicts one of the typical situation in which the system can be found. We have identified authors having papers, each author can have more papers, each paper can have more authors. Furthermore, there are reviewers and one or more papers can be assigned to each of them. Each reviewer creates the reviews, one review for each assigned paper.

The identified objects can be sub-classified to passive objects and active subjects. The passive object has no own activity, it serves particularly as a data storage and allows to manipulate with encapsulated data. The active subjects can offer the same protocol as the passive objects and, moreover, defines its own behaviour which can be processed independently from other objects or subjects. In our example, the author and the reviewer are active subjects, because they manipulate with passive objects (the review and the paper).

B. Roles Identification

The object diagrams primarily serves for object identification and helps to classify them into classes. Before it, we have to identify roles in the system and to distinguish between objects and their roles. For instance, the reviewer and author are both persons and should be represented by one kind of entity. Now we can classify previously found objects of authors and reviewers into the class `Member` whose objects can play the role of *author* or *reviewer* (or both). The paper and the review serve just for data storage so there is no need to identify their roles. The class diagram is shown in the Figure 10.

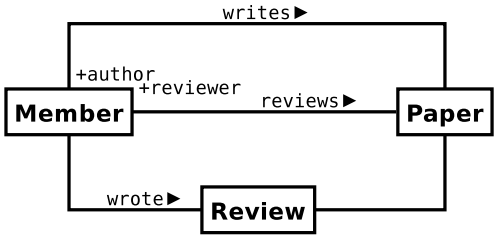

Fig. 10. The class diagram of persistent objects.

So far, we have classified three classes representing three kinds of *persistent objects—paper*, *review*, and *member*. The paper and review represent passive objects while the member is active object whose activities are modelled by its roles.

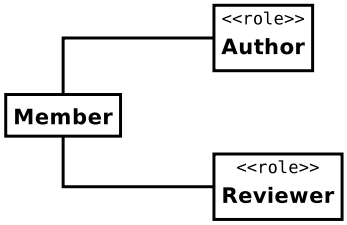

Fig. 11. Classification of roles.

Each role (and, thus, appropriate activity) is modelled by means of the OOPN formalism—to each role there is one class (see chapter VIII-C). The behaviour of appropriate roles are therefore modelled by OOPN objects which wrap the member object. The situation of our example is shown in the Figure 11.

VII. METHODS ACCESSING THE OBJECT STATE

Each object can offer the protocol for setting values, getting values, and determination based on the object state. It can be modelled by two ways—method nets or ports. Both ways have their advantages and disadvantages. We will discuss it in this chapter.

A. Determination Based on the Object State

Using method nets seems to be simpler to understanding and implementing, but there is no possibility to test an object state without calling the method. That means the calling transition has to be fired and followed by structure making decision about fruitfulness of the operation.

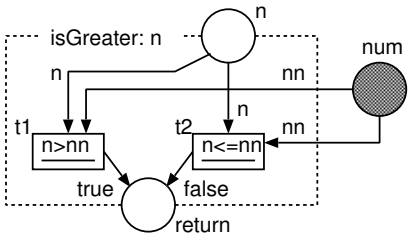

Fig. 12. The method net isGreater:.

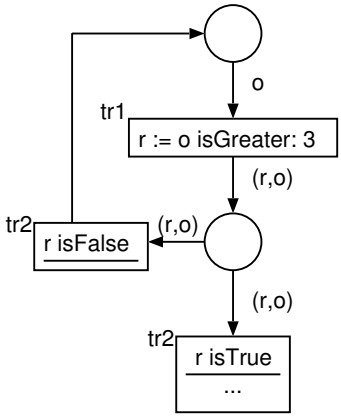

Fig. 13. Using of the method net.

The example of using methods is shown in the Figure 12. There we can see the method *isGreater:* having one argument, the number *n*. The purpose of this method is to decide whether some encapsulated value is greater than *n* (then the method returns value *true*) or not (then the method returns value *true*). The Figure 13 depicts using of the method *isGreater:*. We have to call this method from the transition body (*tr1*) and store the result to the place. Now we have to define two transitions for two different results—true and false.

As we can see, this approach for behaviour determination based on the object state is little bit complicated. The better solution is to use synchronous and negative ports. The Figure 14 shows definition of the port *isGreater:* and the negative

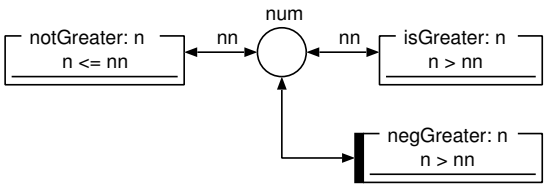

Fig. 14. The synchronous and negative ports.

port *notGreater:*. The Figure 15 shows the using of these ports on the same example. If the number stored in the place *num* is greater than number 3 then the port *isGreater:* is firable and, thus, allows for firing the transition *tr2*. Otherwise, the port *notGreater:* is firable along with the transition *tr1*. For the second variant we can use the negative port *negGreater:* too. There are situation where using of synchronous ports is very difficult and we have to solve it by negative ports. Using of ports for the presented purpose we will call *querying* in this paper.

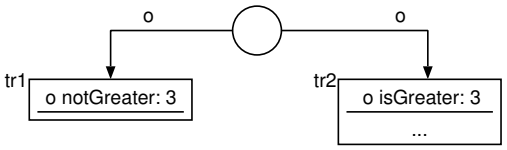

Fig. 15. Using of the synchronous ports.

B. Getting Values

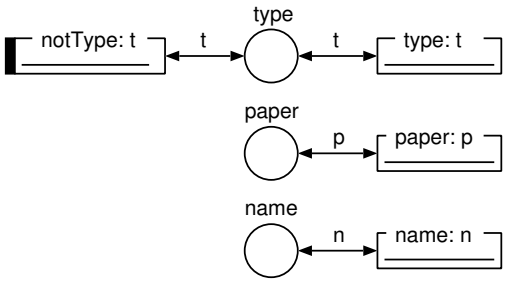

Fig. 16. The member net.

Let us continue in our case study. The object net `Member`[1] is shown in Figure 16 (this class represents persistent objects of type *member*). It contains places for state information (`type` and `name`) and the place for aggregated persistent

[1]Each OOPN class consists of just one object net, so that it is possible to identify this object net by the class name.

objects (`papers`) including synchronous and negative ports. These ports can be used for the state determination, e.g., the synchronous port *name:* can be used for determination whether this object represents the member of passed name. But, it is also possible to use the same synchronous port for getting value (i.e, the name of the member). If we will call this port with unbound variable, this port will be firable and will bind the content of the place *name* to the passed variable (in other hand, if such the port is fired, it returns the member's name).

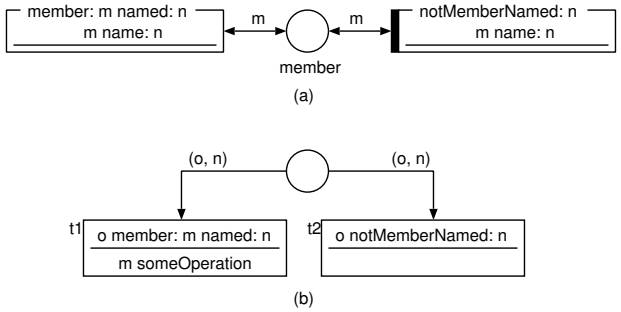

Fig. 17. Getting values.

Of course, there is a possibility to get values using method nets. Consider that we have an object having members (objects of the class `Member`) in the store. The object offers the operation allowing to look for a member with passed name. Using methods, we have to iterate through all members stored in the object's place. The direct iteration consumes time during its modelling and does not make the model more readable. Using ports is much better from this point of view. The port says if this operation is possible or not and, moreover, it is possible to chain port calling. We have to fix a situation when the main operation fails, e.g., if there is no member with passed name. To cover it we keep at disposition negative ports. The example is shown in the Figure 17. The Figure 17a depicts the synchronous port *member:named:* and the negative port *notMemberNamed:* of the object having members in its store. The Figure 17b depicts using of these ports. If there is placed a member named *n* in the object *o* then the transition *t1* is fired. Otherwise, the transition *t2* is fired.

C. Setting Values

The content of places is modifiable by means of method nets. Each such method net is constructed by course of the same template as shown in Figure 18. It is possible to model it by synchronous ports as shown in the Figure 19 too. But, in many cases, the passed value is a result of previous computation and it is usually better to set it using the method net.

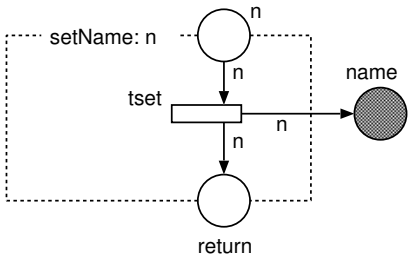

Fig. 18. The method net setName.

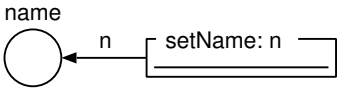

Fig. 19. The synchronous port setName:.

VIII. BEHAVIOUR DEFINITION

The behaviour is defined in two ways which are combined. Firstly, the workflow modelled as a sequence of fired transitions or synchronous ports. Secondly, the state querying modelled by synchronous ports and negative predicates. We will demonstrate these ways in different behaviour nets.

A. The Model of Passive Objects

The passive object of a paper is modelled analogous to the objects described in the previous chapter. The object net `Paper` (see Figure 20) contains information about submitted paper, its review, and its authors including synchronous ports and negative predicate for querying.

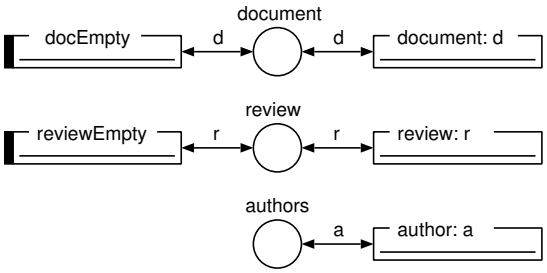

Fig. 20. The paper net.

Some of setting methods differ from analogous methods in the object net `Member`. For example, the method

`putDocument:` (see Figure 21) allows to re-submit a paper, so the old version has to be removed and the new one is put. This is to be taken as the other template of accessing method nets.

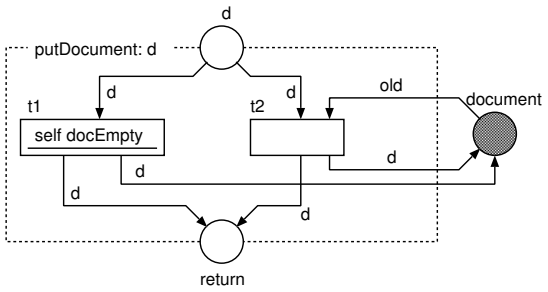

Fig. 21. The method net putDocument:.

B. The Conference Behaviour

At the current time, only some actions are allowed—it depends on the state of the conference system. So we have to add one subject—the conference itself.[2] 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 operations allowed in appropriate states:

- *Open.* The author can submit his paper to the system. He is also able to read the submitted paper and resubmit it again. Other subjects can do nothing.
- *Review.* The chair can assign submitted papers to chosen reviewers. The reviewer can put his review to the system. Other subjects can do nothing.
- *Final.* The author can read the reviews of his papers and (if the paper was accepted) put the final paper. Other subjects can do nothing.
- *Close.* The conference system is closed, everybody can do nothing.

The conference object stores registered authors and reviewers too. The object net `Conference` is shown in Figure 22. 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.

The way of state changing is appreciable now. Of course, in real application the real time or the administrator is the entity that will change the state of the system. Nevertheless, during the development, we can simulate it for the testing purpose. The simulated state changing is depicted in the Figure 22 by the place storing information about changing the state and by the transition *nextPhase* which do the change according to these conditions. For instance, the conference is in the phase *open* (i.e., the symbol *#open* is placed in the place *phase*) immediately. The next phase *review* will be switched after 100 clock units.

[2]As we can see, the basic division of development process into phases does not mean that these phases have to go sequentially, but can be executed concurrently.

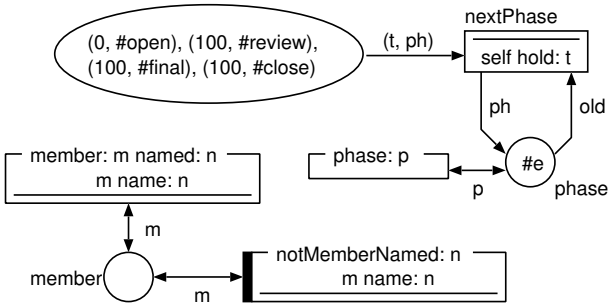

Fig. 22. The conference object net.

C. The Role Behaviour

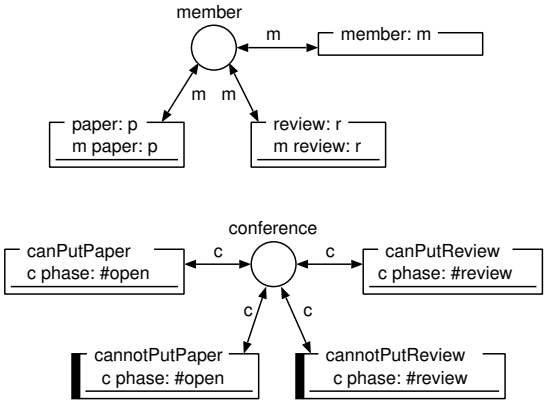

Fig. 23. The author activity net.

We will show the role behaviour definition on the example of the author role. In our example, the author is active only if the states open and final are set. Because we suppose the external access to the system (the real author will manipulate with the system), the author net does not define workflow, but just the state querying as we can see in the object net `AuthorNet` in Figure 23. The object net `AuthorNet` consists of two places, the place `member` containing an object of author, and the place `conference` containing an

object of conference. It defines following synchronous ports and negative predicates:

- `member:` testing if this net is for passed member object or allowing to get the member object
- `paper:` testing if the author net contains passed paper or allowing to get the paper (perhaps even papers, if the author submitted more papers) – this operation is delegated to the member object
- `canPutPaper` testing if it is possible to put paper to the system
- `cannotPutPaper` testing if it is not possible to put paper to the system

Of course, we also have to define a way to modify the author state. It is possible to model it by synchronous ports with second effect or by method nets. The synchronous ports have one disadvantages in its use—if we want to check a result, we have to define negative predicates which are complementary to the ports, while the method nets can return its result directly. The example of the method nets putting the paper of the author to the conference system is shown in Figure 24. This method returns whether the operation was successful or not.

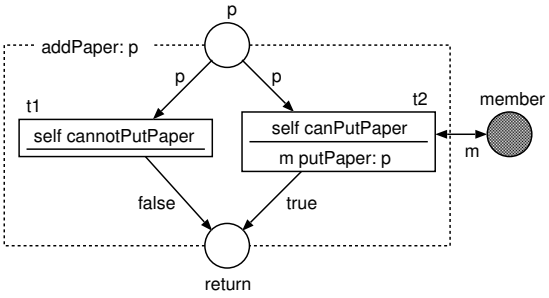

Fig. 24. The method net addPaper:.

How to use synchronous ports for testing is shown in Figure 24, where the possibility of the paper putting is checked. But we can get a collection of appropriate objects by means of synchronous ports as well. We will demonstrate it on the external accessing via meta-object protocol. Let us have a script for testing the model created so far, see Figure 25. The object of `AuthorNet` is accessible by the variable `authorNet`, the object of `Member` is stored in the variable `member`. After we put a member to the author net (by calling `setMember:`), we can check if the member is really placed into the object net. At first, we get a special meta-representation of the object allowing us to work with synchronous ports in more comfortable ways (the message `asPort`). Then we can call synchronous ports similarly to the method with the difference that we get a special object as a result. This result can be tested as boolean variable and, moreover, contains potentially bindings of variables as we can see at the bottom of the script. The synchronous port `paper:` is called with unbound variable named `p`. It means that we want to get all papers associated with the author. The found objects are stored in the result as a collection.

```
authorNet setMember: member.
anp := authorNet asPort.
res := anp member: member.
res ifTrue: [ ... ].
res := anp paper: (PNVariable name: #p).
res ifTrue:
    res do: [:bind | bind at: #p ]
].
```

Fig. 25. Testing script 1.

IX. LAYERED MODELLING OF BEHAVIOUR

The behaviour models can be layered so that the behaviour is defined at different levels, where one level wraps some other level. We will show it on example in this chapter.

A. The Registration Behaviour

Each member must log-in to the system before any action. The process of member identification (we leave out the registration) is modelled as a net which is instantiated whenever the new user connects to the application (see Figure 26). The sequence of activity is *login*, *verify user*, and *logout*. If the user verification is successful, the net describing the user behaviour 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.

The activity is modelled 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 analogous to the methods. The transition models *an internal event* and the synchronous port models *an event synchronised with some external event*. As an example we may take an event *login:as:*, which is modelled 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 the chapter IX-B).

The event *verify* is modelled 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 modelled by the method net `verify:as:` (see

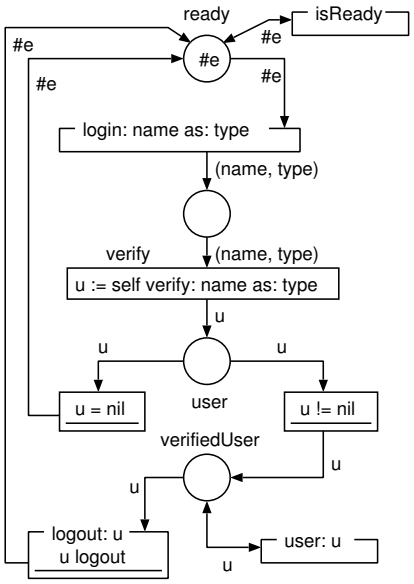

Fig. 26. The registration activity net.

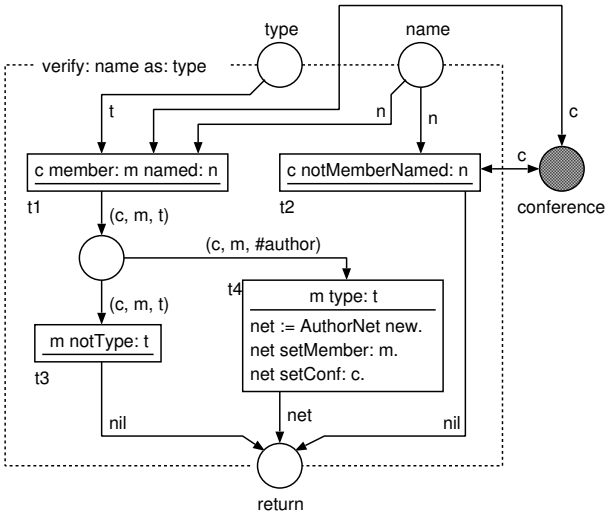

Fig. 27. The method net verify:as:.

Figure 27). If the user verification successes (i.e., there is a member with passed name and having the `#author` type), this net returns a net corresponding to the `AuthorNet`. Otherwise it returns `nil`.

B. The Interface to the Registration Behaviour

To testing registration behaviour we can use two basic ways—to create a testing net or to use meta-object protocol. We will pay attention to the first way now.

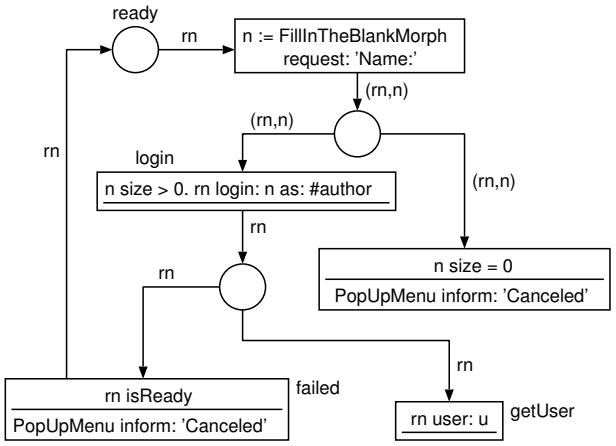

Fig. 28. The user interface net.

The user interface can be modelled as the net shown in Figure 28. 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 26 at the same time.

Now the user verification is being processed (see calling the method *verify: name* in the transition *verify* in Figure 26) and the user interface net (Figure 28) 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 26. If the registration process is successful, the new net representing a user behaviour 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 28).

The second way is to use meta-object protocol outside of the OOPN formalism, i.e. from the Smalltalk environment. The example testing a log-in action by the second way is shown in Figure 29).

```
m := Member new.
m setType: #author. m setName: 'me'.
c := Conference new. c addMember: m.
net := RegistrationNet new.
net setConference: c.
np := net asPort.
res := np login: 'me' as: #author.
res ifTrue: [ res perform ].
```

Fig. 29. Testing script 2.

X. CONCLUSION

The paper has outlined basic principles of the Object Oriented Petri Nets (OOPN) formalism, the associated PNtalk framework, and their usability in the system design. OOPN along with interactive incremental development allows to design systems at different levels of abstraction using sequential reinforcements and improvements. There were demonstrated key modelling techniques needed to OOPN using in the system design.

The PNtalk system has been designed as a tool suitable for system development based on modelling and simulation. Both OOPN formalism and PNtalk system are closely associated with Smalltalk environment. The important advantage concerns a possibility to use OOPN as a part of Smalltalk environment in different scenarios. The developed model can be incorporated with Smalltalk program, so that it is connected to real environment. It allows for safer and more effectively design and testing of software systems—we are able to check correctness by simulation or formal verifications.

The important idea is also *model continuity*. It makes the tendency towards an elimination of generating the source code from models. 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 [11] gets better. 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. 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 presented modelling techniques are part of development methodology designed by our research team. Its characteristic is to use formal models (OOPN) in conjunction with modelling and simulation techniques, combination of modelled and real components, and incremental development process, which allows for high-quality and rapid development. We have already experimented with a simple case studies in the field of software engineering [9] and agent systems [15]. The further research will be aimed at the efficiency of simulation techniques and at the better tool support.

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

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

[3] C. A. Lakos, "From Coloured Petri Nets to Object Petri Nets," in *Proceedings of the Application and Theory of Petri Nets*, vol. 935. Spinger-Verlag, 1995.

[4] C. Sibertin-Blanc, "Cooperative Nets," in *Proceedings of Application and Theory of Petri Nets*, vol. 815. Springer-Verlag, 1994.

[5] 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*, 2005, pp. 148–167. [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3536&spage=148

[6] 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*, vol. 3099. Springer, jun 2004, pp. 484–493. [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3099&spage=484

[7] V. Janoušek, "Modelování objektů Petriho sítěmi," Ph.D. dissertation, FEI VUT, Brno, CZ, 1998.

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

[9] 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*. ARGE Simulation News, 2007, p. 9.

[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*. Hamburg, DE, 2006, pp. 301–307.

[11] 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.

[12] 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*. Las Palmas de Gran Canaria, ES, 1997, pp. 229–231.

[13] V. Janoušek and R. Kočí, "PNtalk: Concurrent Language with MOP," in *Proceedings of the CS&P'2003 Workshop*. Warsaw University, Warsawa, PL, 2003.

[14] 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.

[15] V. Janoušek, R. Kočí, Z. Mazal, and F. Zbořil, "PNagent: a Framework for Modelling BDI Agents using Object Oriented Petri Nets," in *Proceedings of 8th ISDA*. IEEE Computer Society, 2008, pp. 420–425.