PNtalk: Concurrent language with MOP

Radek Kočí Vladimír Janoušek

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

Abstract

The PNtalk is a project comprising concurrent language and system based on object-oriented Petri nets (OOPNs). PNtalk benefits from the features of Petri nets (formal nature, suggestive description of parallelism, theoretical background) as well as object-orientedness (abstraction, encapsulation, inheritance, polymorphism, and modularity). The purpose of PNtalk is to model, to simulate, to verify, and to prototype concurrent and distributed systems, including systems with dynamic structure. Needed flexibility and adaptability is accomplished by metalevel architecture of PNtalk system. This paper presents key properties of PNtalk, its metaobject architecture and its metaobject protocol (MOP).

Keywords: object-oriented Petri nets, open systems, interoperability, Smalltalk, object behavior, meta-object

1 Introduction

The simulation, simulation tools, simulation techniques, and object systems are issue of an interest of our research group at Faculty of Information Technology, Brno University of Technology. One of results of that interest is the project PNtalk. The PNtalk deals with the modeling, simulation, verification, and prototyping of non-trivial software systems. It has been focused on object-oriented Petri nets (OOPNs) allowing a formal, hierarchical, and object-oriented description of modeled problem. The PNtalk system was originally designed as a transparent layer on top of a Smalltalk system.

There have been introduced some experimental implementations of such system and presented some ideas about modeling and simulation using OOPNs for several years. On the basis of existing research and results [3, 1] we have recently focused on architectures of simulation systems which satisfy demands of highly dynamic systems modeling and simulation (e.g. agent systems).

The key idea of our newly developed simulation system is flexibility. We have several goals. Firstly, a possibility to modify the semantics of basic formalism, to add new formalisms, or to combine different kinds of formalisms. Secondly, a possibility to modify the context of processes, to access inner structures and values of objects, or to experiment with special simulation techniques (e.g. nested simulation [7]). Thirdly, a possibility to make up models and to derive prototypes from them in simple way. And also to extend the system to distributed environment.

In present, we have developed the kernel of a system which allows us to apply features connected with our second goal. The first goal will require further research. The third goal is to be resolved in the near time. It is partly related to combination of different paradigms because of the need to connect the basic simulation formalism with others. This concept can partly allow a heterogeneity of formalisms, for instance a combination of OOPNs and classic simulation techniques.

1.1 PNtalk and OOPNs

We can take a look at PNtalk from several points of view. PNtalk is a language for describing models based on OOPNs; it is the system for the simulation, analyze, and verification of those models; it can be the technology of an object-oriented design, modeling, and prototyping. Now, we will try to describe the OOPNs and their features that are important for the viewpoint of the PNtalk system architecture. Our description will reflect OOPNs defined in [3].

PNtalk and the associated OOPN formalism are characterized by a Smalltalk-like object orientation enriched by concurrency and polymorphic transition execution, which allow message sending, waiting for and accepting responses, creating new objects, and performing primitive computations. OOPNs are based on viewing objects as active servers[1] that offer reentrant services to other objects. Services provided by objects as well as independent activities of particular objects are described by means of high-level Petri nets consisting of inscribed places, transitions, and arcs.

When some object calls for some service of another object — i.e. the object (sender) sends a message to another object (receiver) — a new process is created in the receiver. This new process can be understood as the instance of the service description (in object-oriented terminology, the method is invocated). The process is characterized by the sequence of the transition executions. If some transition sends a message to another object, then it has to wait for response. i.e. the processing of this transition is suspended[2]. As it results from the Petri nets definition, each process can have several transitions that are independent each other, and consequently they can be executed concurrently. Thus, there is a parallelism even within the OOPNs process. It also implies the object life has to be evaluated rather by events, not like a sequential process, with understanding the event as a firable transition.

2 PNtalk System

In addition to many other things (such as a possibility to modify the context of objects, to access inner structures etc. – q.v. the chapter 1) we ask for full controllable objects too. It means we would be able to start the object, to stop it, to store it, to restore it, and so on. Due to those requests the whole object is controlled by some Smalltalk process providing the operation processing. It follows the message passing between objects must be replaced by delegation, sequential processing is replaced by event processing, and the classic object-oriented approach is to be adapted to metaobject-oriented approach. On the basic layer there are metaobjects representing and controlling the domain objects in the system. Hence, we need to distinguish two kinds of operations:

domain protocol defines operations offered by the domain object (i.e. object living in the user's world which describes solved problem). The domain objects are being made by users during life of the system, thus the domain protocols are compiled dynamically.

meta protocol defines operations needed by metaobject to manipulate both its domain object and the metaobject. The meta-objects are made as fixed components in the system, thus the metaprotocol is made as fixed and it is known before starting the PNtalk system. Of course, it may evolve thanks to the underlying Smalltalk system.

[1] The active server means the object can have own activity which starts as soon as the object is created (it is called the *object net* in OOPNs).

[2] Take a notice the message sending is rather the request for message processing by the receiver and the receiver processes the message within a newly created process.

There is a difference among the operations of metaprotocol consisting in influence on the component. For that reason we divide the meta-operations into some categories by the influence on the component. We can found three basic categories:

- *query operations* serve only for information getting and do not modify the context of the component. The system need not guarantee the mutual exclusion of these operations.
- *modify operations* serve for modification of the context of the component. The system must guarantee the mutual exclusion of these operations.
- *control operations* serve for the object life controlling. The system must guarantee the mutual exclusion of these operations.

Objects communicate each other by means of the message passing. So if an object sends a message to another object then it has to wait for method response. Moreover, we work with the OOPNs (thus the invocated method consists of high-level Petri nets and tokens are represented by the object references) what implies two things. The object is rather the actor and there is a parallelism inside the OOPNs objects. Although we talk about the invocated method like the process, this process can fork more subprocesses (and, of course, it can send messages simultaneously) in dependency on the net evolution. Due to requirements on the system the live object does not correspond with the OOPNs classes directly. Consequently the process of object, which has invocated some method, cannot wait for method response but it must be able to continue in event processing. Hence the message passing with synchronous messages is replaced by asynchronous meta messages with ensuring the relationship between invocated method and transition which have sent the message. This relationship allows the transition to get information about method finishing and to get the result of the method.

There also arise some synchronization problems – if more than one process sends a message to the object A then the invocated methods of the object A can modify this object A simultaneously. It follows there must be some synchronization means to mutual exclusion of simultaneous influence of processes. It does not only concern operations from the domain protocol but it also concerns operation from the metaprotocol. The synchronization is ensured by so-called serialization of messages – it means all appropriate received messages are placed into a shared queue and the process of metaobject then process those messages one by one from the queue.

2.1 Interoperability

The above sketched architecture (metaobjects, asynchronous messages, etc.) is not able to communicate with its envitonment (Smalltalk) seamlesly. It offers only low-level communication means (e.g. we must know how to send a domain message) which is not compatible with other systems (especilly with Smalltalk system). Nevertheless, there can arise the situations when some object (sender) living outside the system need to communicate with an object (receiver) living inside the system. Ideally, the sender does need to not know the kind of the receiver.[3] There is a need of a means for transparent communication (the interoperability) between objects of different types, and for communication with objects on different levels (the domain level or the meta level). Hence we introduce new special metaobject enabling interoperability and special access to objects – so-called proxy-objects.

Using proxy is standard Smalltalk technique to control message passing. A proxy is obviously implemented in such a way that it handles the exception *messageNotUnderstood*. The handler (implemented in the proxy) decides how to react to the message.

[3] An object can communicate with any objects (receivers) by the same form and the receivers have to understand this form.

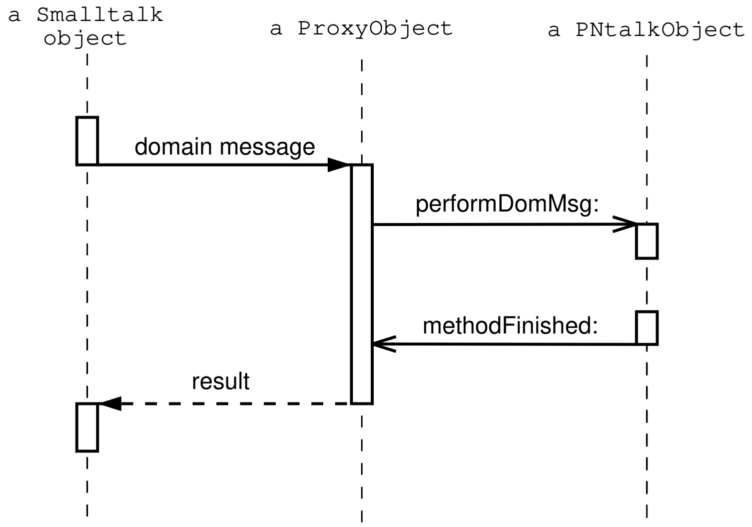

Figure 1: The Proxy object

The basic overview of proxy using is shown in the Figure 1. An object (potential sender of a message) always refers to another object (potential receiver) by means of a proxy-object delivering incoming messages to the receiver in the suitable form. There are several kinds of proxy-objects allowing the communication at the domain level or at the meta level. The behavior of the proxy-object also depends on a kind of the sender and the receiver.

The present concept of meta-objects and proxy-objects together represent the basis of distributed implementation (i.e. collaborating of far objects including the possibility of object migration). Nevertheless, there are still several open questions for future research in this area.

2.2 The system architecture

This chapter briefly presents an architecture of the simulation system. We can see several levels of activity in the system. Firstly, we have a description of models (or parts of models) – a source code. It defines classes of OOPNs. Then we get a model representation from this description as a result of (incremental) translation. The model representation consists of components determining OOPNs classes and defines the domain behavior (messages) of object. The classes are represented by the instances of *PNtalkClass* including appropriate sub-components (nets etc.). This level is structured as a tree with the special component *PNtalk* as a root. The component *PNtalk* represents a basic name space consisting of OOPN classes (*PNtalkClass*) or another name spaces. For the sake of simplicity we can suppose there is only the top-level namespace in the system.

The PNtalk objects are at the domain level instances of PNtalk classes. At the implementation level they are instances of *PNtalkObject* and they comprise appropriate sub-components (processes etc.). The instances of OOPNs classes (thus instances of metaobject *PNtalkObject*) are always placed in some world and each world is included into the so-called processor. That structure is shown in the Figure 2. We can see there following components:

- *PNtalkObject* representing the OOPNs object and providing means for processing domain protocol.
- *PNtalkWorld* representing a group of OOPNs object collaborating in the simulation. There always exists at least one (default) world.

- *PNtalkProcessor* representing the system of OOPNs worlds.

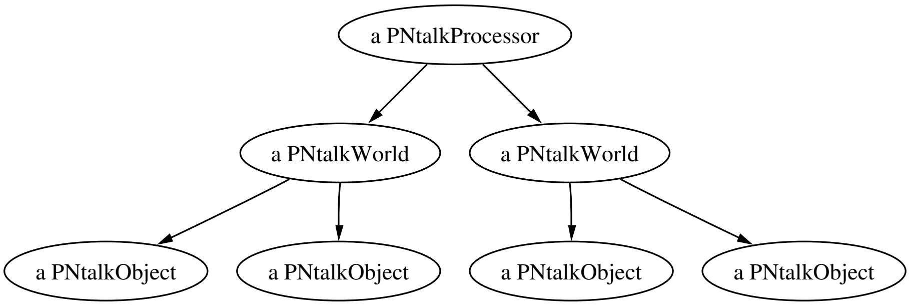

Figure 2: The PNtalk architecture - basic overview

The Figure 3 presents overview about a structure of the object representation (*PNtalkObject*) and class representation (*PNtalkClass*). Both components consist of nets with having a little difference – the sub-component *PNtalkCompiledNet* represents compiled method of the OOPNs object while the sub-component *PNtalkProcess* represents invocated method (e.g. the instance of the net). The sub-components *PNtalkCompiledNet* and *PNtalkProcess* consist of transitions and places with having a little difference again. The transitions (*PNtalkTransition*) and places (*PNtalkPlace*) are instances of transitions (*PNtalkCompiledTransition*) and places (*PNtalkCompiledPlace*) and they have their marking. The marking of place comprises object references, the marking of transitions comprises its invocation (i.e. the fired transition). The invocation of transition is represented by the component *PNtalkTransitionInstance*.

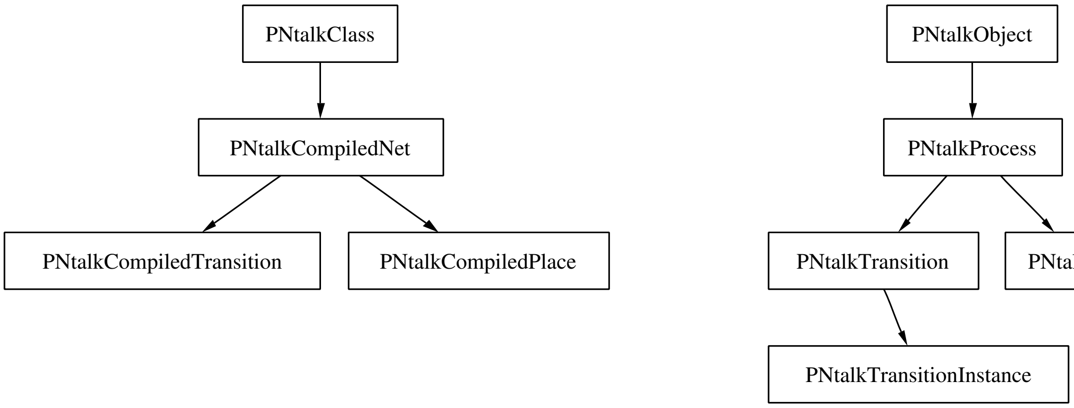

Figure 3: The structure of *PNtalkClass* and *PNObject* - basic overview

There is one exception in viewing the metaobjects. All of them serve as a structural definition of the system object or the domain objects (*PNtalkWorld*, *PNtalkTransition*, etc.) and only *PNtalkObject* serves as a representation of a *domain object* and therefore only *PNtalkObject* offers metaprotocol concerning the domain object behavior. The metaprotocol is hidden from the domain object's point of view. It can be accessed via the proxy-object.

The metaobject processes all its events by a step-by-step manner. The object may do next step only if its metaobject receives appropriate control message (e.g. *step*).

3 Metaobject protocol

In this section we will turn our attention to protocol of metaobjects (components) presented above. The protocol will be shown and explained on examples. As the model example of OOPN we use a part of the Bank model from [9]. The class *BankModel* (see Fig. 4) contains the object net describing arrival of customers (generated by transition generator) to the bank. The customers are served in parallel by several clerks, which are available in place clerks (it is simulated by parallel invocations of the transition service). Some statistical information is collected in the place stat. The model is designed to be simulated in three possible modes: busy, idle, and very busy. A number 1, 2 or 3 located in a place period represents the mode.

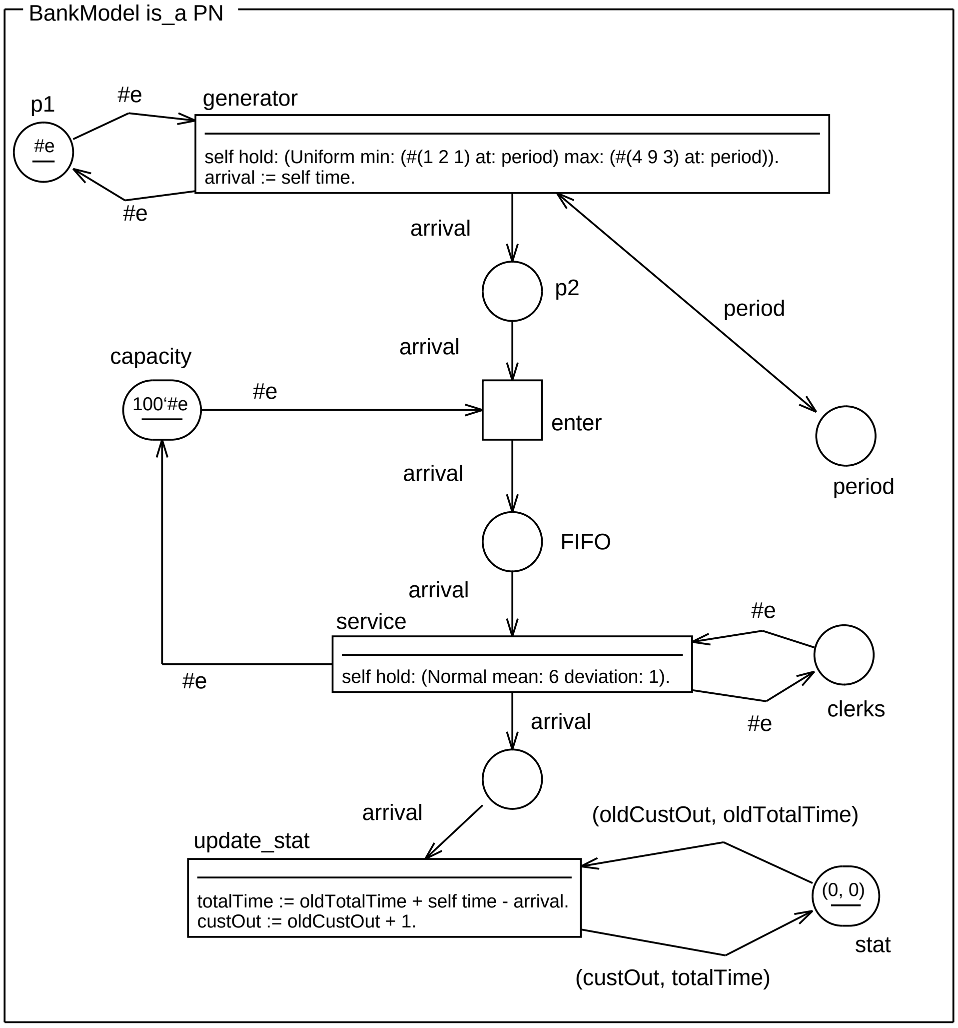

Figure 4: The class BankModel – object net

3.1 The basic protocol

All the basic metaobjects mentioned above share basic protocol ensuring accessing and modification of their hierarchical structure. The metaobjects are represented by instances of appropriate Smalltalk classes. Due to this fact we keep at disposition two groups of protocol – the *instance* protocol and the *class* protocol. The instance protocol serves as protocol of a metaobject while the class protocol serves as a means to metaobject creating or to accessing public informations and components. The basic protocol consists of several operations (the term *receiver* marks the component, which is an actor of operation):

instance protocol	
name	returns receiver's identification (either symbol or number)
listComponents	returns the collection of identifications of receiver's subcomponents
componentNamed:	returns the component which is found by the identification
addComponent:	adds the specified component to the receiver
removeComponentNamed:	removes the specified component which is found by the identification from the receiver

Each component has its unique identification in the system and all access protocols are provided via that identification. Each receiver consists of or controls different kind of component (see Fig. 3). The *PNtalkClass* consists of compiled nets, the *PNtalkProcessor* controls PNtalk worlds, the *PNtalkWorld* consists of (and controls) PNtalk objects, the *PNtalkObject* consists of processes, and the *PNtalkProcess* consists of places, transitions, and transition instances. All these components implement the same interface (metaprotocol) and thus the principle of the basic metaobject manipulation is uniform. Nevertheless, some components may define more specific protocols.

3.2 PNtalkClass

Instances of *PNtalkClass* represent classes of OOPNs. These metaobjects including all subcomponents (instances of *PNtalkCompiledNet* etc.) can be created from the OOPN class descriptions (source texts) by a compiler (another metaobject). PNtalk classes, together with their subcomponents, serve for creating PNtalk objects consisting of PNtalk processes.

class protocol	
newNamed:superClass:	creates new PNtalk class (an instance of *PNtalkClass*)
named:	returns an appropriate PNtalk class according to a name

instance protocol	
compile:	compiles a PNtalk class source
compileNet:	incrementally compiles a method net or object net source
new	creates a PNtalk object (an instance of *PNtalkObject*)
newIn:	creates a PNtalk object in specified world

Example 1 *Making OOPN classes and objects*
The following is an example of a text representation of a part of the model shown in the Figure 4:

```
object
   place period()
   place capacity(100`#e)
   place p1(#e)
   place p2()
```

```
trans generator
  cond period( period )
  precond p1(#e)
  action {
    self hold:  (Uniform min:  (#(1 2 1) at:  period)
     max:  (#(4 9 3) at:  period)).
    arrival := self time.}
  postcond p1(#e), p2(arrival)
```

This code is compiled into a class by means of operation *compileNet:* with this code as a parameter. System representation of our model can be created by executing the following piece of code (assuming the above text is referenced by variable *source*):

$$(PNtalkClass\, newNamed:\ \#BankModel\; superClass:\ \#PN)\; compileNet:\ source.$$

An instance of *BankModel* can be created by executing:

$$obj\ :=\ (PNtalkClass\, named:\ \#BankModel)\; new.$$

□

3.3 PNtalkProcessor, PNtalkWorld, and PNtalkObject

Each domain object (or more precisely, its metaobject – *PNtalkObject*) is placed in the system hierarchy. The object always lives in a context of some world which determines (make decision about) its evolution steps. That evolution can be provided in two basic modes – debugging and running. In the running mode the world automatically sends a control message *step* to all its objects (i.e. objects participating in this world). The object chooses one event (a firable transition) to performing in dependence on the message *step*. The world waits for performing the evolution step of an object prior to sending next message *step*. In that way there is ensured continuous run of the world (its objects). In the debugging mode the world does not continually send a control message *step*, but the objects are ready to receive control messages by its neighbourhood (e.g. by the user via Smalltalk environment). The interesting feature is a possibility to switch any component into different modes independently of other components on the same or higher level. For example, if the world is running and one of its objects is being turned into debug mode now, then this object is excluded from the world evolution. When the world is being turned into running mode then all its components are turned into running mode. That relationship between the *world* and objects is the same as the relationship between *processor* and *world*.

PNtalkProcessor and PNtalkWorld These components define further operations allowing user (or another component) to control the life of components. In the first instance the protocol must ensure a simulation run of the model. It provides following control operations:

instance protocol	
test	Ensures testing all components included in the receiver. This operation has solely point in the case of debug mode.
step	Ensures stepping all components included in the receiver. This operation has solely point in the case of debug mode.
resume	Ensures continuous run of all its components.
suspend	Turns all its components into debug mode.

In addition to these control operations, a *PNtalkProcessor* can be asked for default world (operation *defaultWorld*).

PNtalkObject The component *PNtalkObject* offers the same control operations presented above with a bit different semantics. The operations *test* and *step* applied on *PNtalkObject* perform one test (i.e. the object simulator tests all transitions which of them are firable) or one step (i.e. the object simulator fires one of firable transitions). On the other hand the operations *resume* and *suspend* solely produce the *PNtalkObject* gets a sign "I'm active" or "I'm not active". Because of the object life is controlled by environment so this one uses that sign to make decision if the object can do next step or cannot.

In addition to that the component *PNtalkObject* adds the operation *step:* with one parameter determining the transition to fire.

Example 2
Let us continue with our example. We can use the metaobject protocol to modify state of the object obj. As the Figure 5 shows, the object activity depends on places *period* and *clerks*. Therefore we add appropriate objects to these places:

$$((obj\ meta\ componentNamed:\ \#object)\ componentNamed:\ \#period)\ add:\ 1.$$
$$((obj\ meta\ componentNamed:\ \#object)\ componentNamed:\ \#clerks)\ add:\ 10.$$

This is only simple example where it is possible to access dynamic runtime metaobjects by static names derived from compile-time names.

Previous operations ensure that the model will be simulated for period number 1 and for 10 clerks. Due to our explication we will suppose the default world is suspended. The following operations complete one step of object:

$$obj\ step.$$

In a similar way it is possible to step also on the level of *PNtalk world* or *PNtalk processor*. This way we can debug PNtalk models. To inspect content of places (e.g the place *stat*) we can complete the following:

$$((obj\ meta\ componentNamed:\ \#object)\ componentNamed:\ \#stat)\ printString.$$

We can let the model run by executing

$$PNtalkProcessor\ defaultWorld\ resume$$

At any time, it is possible to suspend the simulation, for example:

$$PNtalkProcessor\ defaultWorld\ suspend.$$

□

Example 3
It is also possible to modify behavior of objects at runtime. The following code defines method $experiment$ (see Fig. 4). This method lets the simulation run for some amount of simulation time and then it returns gained statistical information.

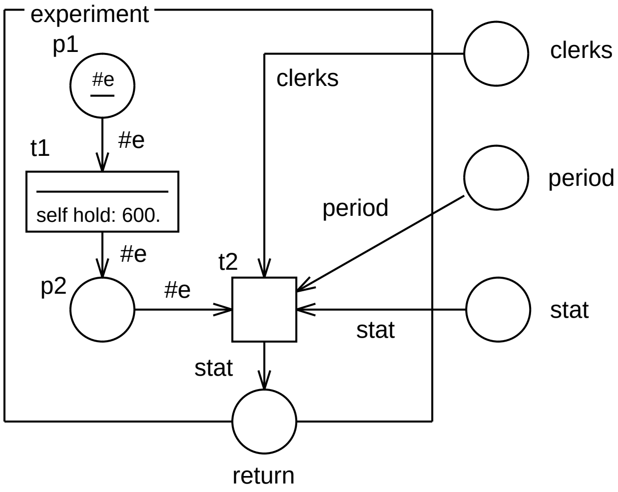

Figure 5: Method $experiment$.

```
method experiment
   place p(#e)
   place p2()
   place return()
   trans t
     precond p(#e)
     action {self hold:  600}
     postcond p2(#e)
   trans t2
     precond period(period), clerks(clerks), stat(stat), p2(#e)
     postcond return(stat)
```

Now we can incrementally modify the class $BankModel$ (let us assume that the method $experiment$ is assigned to variable methodSource):

$$(PNtalkClass\, named:\ \#BankModel)\ compileNet:\ methodSource.$$

After resuming the model run, we can send message experiment to obj.

$$PNtalkProcessor\ defaultWorld\, resume.$$
$$statistics\ :=\ obj\ experiment.$$

□

3.4 PNtalkProcess, PNtalkTransition, PNtalkTransitionInstance

PNtalkProcess Instances of *PNtalkProcess* implement instances of high-level Petri nets (PNtalk methods). Transitions of a net instance can be conditioned by places from this net instance or from the appropriate instance of *object net*. This component offers following further operations (these ones are special cases of operation *listComponents*):

instance protocol	
listPlaces	returns a collection of identifications of places
listTransitions	returns a collection of identifications of transitions
listFirableTransitions	returns a collection of identifications of firable transitions
listPendingTransitionInstances	returns a collection of transition instances waiting for completing their actions

PNtalkTransition A transition can be firable for more different mapping variables on objects. Such a mapping is called the *variablesBinding*. The component offers these further operations:

instance protocol	
listBindings	gets a collection of bindings
fire	fires the transition. If the transition sends a message (i.e. it is non-atomic) then the transition instance is being created.
fire:	fires the transition for some specific variables binding

PNtalkTransitionInstance The transition instance may consist of more actions (message sending) and it has always one variables binding. The operation *listVariables* returns variables binding. The transition instance is evaluated step-by-step, each next step (evaluation of next action) is performed by means of the operation *fire*.

4 Conclusions

In this paper, we described the motivation and basic ideas of the PNtalk system. PNtalk is the open simulation and development environment focused primarily on object-oriented Petri nets. We have introduced the system meta-level architecture and relationships among the meta-objects and their domain objects. The important feature of that architecture is the metaobject protocol. The interoperability between PNtalk and Smalltalk was mentioned too. Thanks to the PNtalk-Smalltalk seamless interoperability there is possible to access the metaobject protocol (MOP) from PNtalk programs themselves, so that reflective programming is possible (models can introspect themselves, modify their state and behavior, and so on).

This work has been done within the project CEZ:J22/98: 262200012 “Research in Information and Control Systems” and also supported by the Grant Agency of Czech Republic grants No. 102/01/1485 “Environment for Development, Modelling, and Application of Heterogeneous Systems”.

References

[1] M. Češka, V. Janoušek, and T. Vojnar: *PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling*. In F. Pichler and R. Moreno-Díaz, editors, *Computer Aided Systems Theory and Technology – EUROCAST’97*, volume 1333 of *Lecture Notes in Computer Science*, pages 591–610, Las Palmas de Gran Canaria, Spain, February 1997. Springer-Verlag.

[2] V. Janoušek: *Reflective Approach to Petri Net Simulation.* MOSIS’97, MARQ Ostrava, pages 299–304.

[3] V. Janoušek: *Modelování objektů Petriho sítěmi (in czech)*. Ph.D. thesis at DCSE FEECS TU of Brno, 1998.

[4] R. Kočí: *Rozšiřitelný simulátor objektově orientovaných Petriho sítí (in czech)*. Tech. Report. DCSE FEECS TU of Brno, 2000.

[5] R. Kočí, T. Vojnar: *A PNtalk-based Model of a Cooperative Editor*. In Proceedings of the 35th Spring International Conference on Modelling and Simulation of Systems – MOSIS 2001, Hradec nad Moravici, Czech Republic, CZ, MARQ, 2001, p. 165-172, ISBN 80-85988-57-7

[6] R. Kočí: *The PNtalk System - a Technique for Object Oriented Modelling*. In: Proceedings of XXIIIrd International Autumn Colloquium, Ostrava, CZ, MARQ, 2001, p. 151-158, ISBN 80-85988-61-5

[7] E. Kindler et al.: *Special Approach to Reflective Simulation*. In: Proceedings of MOSIS 2001. MARQ. 2001.

[8] R. Kočí, Z. Rábová: *The PNtalk System and Interoperability*. In: Proceedings of International Conference MOSIS '02, Ostrava, CZ, MARQ, 2002, p. 73-80, ISBN 80-85988-71-2

[9] V. Janoušek, R. Kočí: *PNtalk - An Open System for Prototyping and Simulation*. In: Proceedings of The 28th ASU Conference, Brno, CZ, FIT VUT, 2002, p. 133-146, ISSN 1102-593X

[10] Yasuhiko Yokote, *The Apertos reflective operating system: The concept and its implementation*, Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), SIGPLAN Notices vol. 27, number 10, ACM Press, New York pp. 414–434, 1992.

[11] T. Ledoux, *Implementing Proxy Objects in a Reflective ORB*, Workshop CORBA Implementation, Use and Evaluation at the 11th European Conference on Object-Oriented Programming (ECOOP'97), 1997.

[12] Brendan Gowing and Vinny Cahill, *Making Meta-Object Protocols Practical for Operating Systems*, 4^{th} International Workshop on Object Orientation in Operating Systems, Lund, Sweden, pp. 52–55, 1995.