PNtalk Project – Current Reasearch Directions

Vladimír Janoušek,[1] Radek Kočí[1]

[1]Brno University of Technology, Faculty of Information Technology,
Božetěchova 2, 612 66, Brno, Czech Republic
{janousek,koci}@fit.vutbr.cz
http://www.fit.vutbr.cz

ABSTRACT

PNtalk is a language and system which combines Petri nets and Smalltalk in a consistent way by introducing Object Oriented Petri Nets. The objective of PNtalk is modeling, simulation, and prototyping. Our current focus is heterogeneous modeling, interoperability and multisimulation. To achieve these goals, an open implementation approach proposing appropriate level of control over selected features of models is adopted. The open implementation of PNtalk is accomplished by its reflective metalevel architecture. Multisimulation is demonstrated by a simple case study.

KEYWORDS

Metalevel architecture, Reflectivity, Modelling, Simulation, Multisimulation, Prototyping, Object Oriented Petri Nets

1 INTRODUCTION

PNtalk is a long term project leading towards a tool for modelling, simulation, and prototyping complex systems. As the primary formalism of the PNtalk system, the Object Oriented Petri Nets (OOPNs) have been chosen. OOPNs [7] combine advantages of object orientation and Petri nets. The OOPNs basis consists of high level Petri nets allowing to describe the work-flow of models and their parallelism. Object orientation brings mainly the structuralization of models and communication protocol between parts of the model (objects). OOPNs define objects in a very similar way like other object oriented languages but with one important difference – methods are not described as sequences of commands but by means of high-level Petri nets. At a method call, a new instance (a copy) of the appropriate net is created and made ready for running.

Our current focus is a higher level of dynamism and a higher level of control over the PNtalk features. PNtalk classes are special objects which can be built incrementally during run time (like in Smalltalk [5]). A method is then understood as the least unit constituting the basic block of the class. Along with considering the method to be a pattern and the invoked method to be a copy of that pattern, we can also think about dynamic changes of those structures. By the pattern change, we gain new behavior of its new copies (invocations). The copies originated till this time are not changed. Nevertheless, the copy itself can be modified according to the same principle. The dynamic changes are not restricted just to Petri nets modifications – a possibility to modify or to replace the used paradigm (modeling language) can be very interesting, e.g. by other variant of Petri nets or directly by Smalltalk.

To achieve the above sketched goal the open implementation issue including the reflective and metalevels architectures, appear to be a possible and good way. The feasibility of open approaches depends on the degree of their implementation (thus on what the designers do consider as a profitable limit of the open implementation degree). The PNtalk system architecture

is based on the idea of having a system as open as possible. The goal of this paper is to discuss advantages and some possibilities of such an idea.

2 OPEN IMPLEMENTATIONS

In this section, we try to outline and explain the open implementation approach from several points of view. The most important ones are these: why we deal with open implementations and how to make a framework intended for building the open systems.

Recent trend in modern systems for complex application support (e.g. operating systems) is not only to allow applications to use services offered by the given system, but also to offer means to control how these services are provided and processed. This trend can seem to be contrary to the more traditional approach – the black-box abstraction. It says some abstraction (object) should expose its functionality but hide its implementation. The black-box abstraction has many attractive qualities and brings a possibility of portability, reusing or simplicity of the design process. Nevertheless, it does not allow to adapt parts of the system according to changing requirements, to develop applications during its life-time etc. The open implementation principles offer a solution of the problems above. It is needed to remark that these principles should be understood rather like a framework intended for more flexible design and use of black-boxes.[1]

The basic idea of an open implementation is to allow a model to inspect inner aspects of objects (introspection), and possibly affect some (eventually all) those aspects (reflection). The classic case of an open implementation is the metalevel architecture partitioning a model into two layers – the basic (or domain) level and the metalevel [15]. All objects describing the domain problem represent the basic level. To each object at the basic level there is a special object (or a set of objects) at the metalevel – metaobject. The metalevel[2] represents a higher sphere of the control and de facto describes information about information. A metaobject offers the protocol for inspecting and changing selected aspects of its basic object. We can say the metalevel objects control the life-time of the basic level objects. In general, there can be more metalevels – each superior level controls some aspects of its sublevel. The meta-level architecture allows not only to affect structures of objects but also to modify their computational behavior, e.g. the way how the objects react to messages, what other operations are to be processed in a consequence of sending or receiving messages, etc.

As an example, we may mention the Smalltalk system [5]. Each class has its metaclass which has the same meaning as the class for regular objects. Metaclasses control the class behavior and classes control the object behavior. Thus, Smalltalk has two levels of control (it implies there is one metalevel) – the one for object controlling and the one for class controlling. A deeper introduction to open implementations and metalevel architectures can be found in [15, 18, 11, 16].

3 OBJECT ORIENTED PETRI NETS

General ideas about Petri nets have been developed by C.A.Petri in early sixties. Longtime experiences have confirmed that the Petri nets are powerful graphical tools having a mathematical foundation suitable for describing systems and their processes that can be termed as

[1] Thus, it is not a good idea to reprobate the black-box abstraction. However, in some cases it is useful to have mechanisms to partly evade the black-box concepts.

[2] The word *meta* comes from Greek and means something like *after* or *behalf*. In our conception it can be understood as a denotation of something what stays behind an object and reflects (or describes) its features and properties.

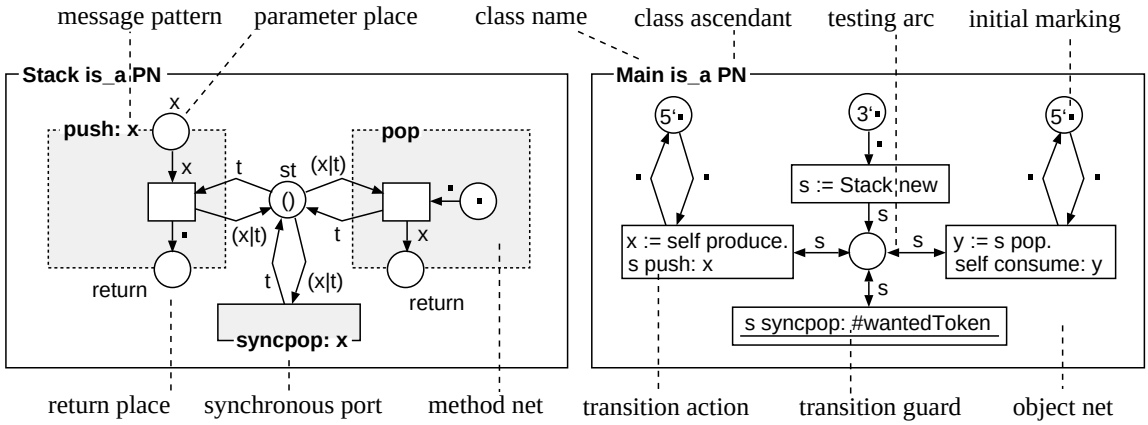

Figure 1. An OOPN example (*Main*'s methods *produce* and *consume* are not shown).

asynchronous parallel and distributed. Nevertheless, Petri nets have some disadvantages. The important one is a lack of lucidity if a model is large. Therefore, many modifications of basic Petri nets have been developed. One type comprises high-level Petri nets. The most well known versions of these are Predicate-Transition nets and Colored Petri Nets (CPN) [10]. The later case is very popular thanks to well-known computer tool based on that formalism. Kurt Jensen, developer of CPN, says that all kinds of high-level Petri nets can be considered as dialects of CPN. The tokens in such nets are colored, which means that they carry some data. An inscription language specifies conditions for transition execution and the effect of the execution. Apart from the high-levelness of some kinds of Petri nets, there has also been some attempts to incorporate modularity and compositionally into Petri nets. Hierarchical Petri nets have been invented and also many variants of object-oriented Petri nets. One such approach is used in the case of PNtalk language and its object-oriented Petri nets (OOPN) [6, **?**]. One type of Petri net that we are merging with object-orientation is very close to predicate/transition nets - all computation is concentrated to the transitions (in our case, a transition inscription comprises a guard and an action), arcs are inscribed by simple expressions (tuples or lists of variables and constants).

Object-orientation of PNtalk and the associated OOPN formalism is based on the well-known, class-based approach in Smalltalk-like style. It means that all objects are instances of classes, every computation is realized by message sending, and variables can contain references to objects. A class defines behavior of its instances as a set of methods (they specify reactions to received messages). A class is defined incrementally, as a subclass of some existing class. In OOPN, this classical kind of object-orientation is enriched by concurrency. Concurrency of OOPN is accomplished by viewing objects as active servers. They offer reentrant services to other objects and at the same time they can perform their own independent activities. Services provided by the objects as well as the independent activities of the objects are described by means of high-level Petri nets - services by method nets, object activities by object nets. Tokens in nets are references to objects. Apart from the concurrency of particular nets, the finest grains of concurrency in OOPN are the transitions themselves (they represent concurrency inside a method or object net).

An example illustrating the OOPN formalism is shown in Figure 1. As it is depicted in Figure 1, a place can be inscribed by an initial marking (a multiset of objects) and an initial action (allowing a creation and initialization of complex objects to be initially stored in the place; not shown in the Figure 1). A transition can have a guard restricting its firability and an action to be performed whenever the transition is fired. Finally, arcs are inscribed by multiset expressions specifying multisets of tokens to be moved from/to certain places by the arcs associated with a transition being fired.

The OOPN on the Figure 1 demonstrates that the method nets of a given class can share access to the appropriate object net – the places of the object net are accessible from transitions belonging to a method nets. In this way the execution of methods can modify the state of the object. The class Main describes an active object, which can instantiate (and communicate with) a passive object - stack (an object is passive if its object net contains no transitions). Each method net has parameter places and a return place. These places are used for passing data (references to objects) between calling transition and the method net. Apart from method nets, classes can also define special methods called synchronous ports that allow for synchronous interactions of objects. This form of communication (together with execution of the appropriate transition and synchronous port) is possible when the calling transition (which calls a synchronous port from its guard) and the called synchronous port are executable simultaneously.

The transition guards and actions can send messages to objects. An object can be either primitive (such as number or string - defined by Smalltalk), or non-primitive (defined by Petri nets). The way how transitions are executed depens on transition actions. A message that is sent to a primitive object is evaluated atomically (thus the transition is also executed as a single event), contrary to a message that is sent to a non-primitive object. In the latter case, the input part of the transition is performed and, at the same time, the transition sends the message. Then it waits for the result. When the result is available, the output part of the transition can be performed. In the case of the transition guard, the message sending has to be evaluable atomically. Thus, the message sending is restricted only to primitive objects, or to non-primitive objects with appropriate synchronous ports.

3.1 Interoperability

One of the main motivations behind the development of OOPN/PNtalk was a possibility to use Petri nets as a part of a Smalltalk program [6]. PNtalk objects can be directly available in Smalltalk, and so an arbitrary Smalltalk object can send messages to the PNtalk objects as though they were Smalltalk objects. Nevertheless, it is necessary to deal with concurrency of OOPN and Smalltalk threads. When a Smalltalk object sends a message to a PNtalk object, a Smalltalk thread is blocked until the invoked method net is finished. That is OK but the opposite case is a bit problematic.

Generally, the execution of Smalltalk code inside a transition action can be done either atomically or non-atomically. It has to be decided at run-time. Since such a decision is not a trivial problem, some single threaded implementations of the PNtalk interpreter allow only atomic execution of methods of Smalltalk objects. If it blocks, this is considered a fault because it blocks the OOPN interpreter. It is up to the programmer to avoid such situations. A multithreaded interpreter (a thread for each transition) works fine but it is much less controllable then we need e.g. for debugging. Our current PNtalk design employs reflective features of Smalltalk-80 in order to allow full control over simulation as well as the ability to communicate with smalltalk's objects atomically as well as non-atomically. In our implementation, an atomic interaction takes place only in the case of a communication with a constant or an instance of some standard classes (e.g. containers). Otherwise, the interaction is performed non-atomically, i. e. as a sequence of two atomic actions corresponding to sending a message and receiving its result.

Note that the above discussed client server interaction mechanism is potentially suitable for interoperability of OOPN with an arbitrary object oriented language and it could be made to conform to CORBA or RMI (i.e. it could be potentially suitable for distributed programming).

3.2 Time in OOPN

An object can run in real time as well as in simulation time. An execution of a transition can be delayed in real time for example by waiting for a response from the user (this can be

accomplished by sending a message to a user interface). Meanwhile, other transitions can be executed if they are executable. A scheduler maintains execution of transitions. A transition can also invoke a real time delay by sending the message *wait* to an instance of standard Smalltalk class *Delay*. Another possibility for a transition to be delayed is waiting for a semaphore.

In a similar way it is possible to specify delayed transition execution in simulation time. Similarly to Simula, the delay is accomplished by sending *hold:* to *self*. To make it possible, an object has to be attached to some simulation scheduler which is capable of maintaining simulation time. In the current PNtalk design, each object is attached to its default scheduler.

Note that apart from transitions with delays, it is possible to introduce time to the Petri nets in some another way (e.g. timed transitions waiting for specified enabling time, timed places, etc.). Nevertheless, the transitions with delays are sufficient enough for simulation tasks like the one demonstrated here.

4 PNTALK SYSTEM ARCHITECTURE

A basic overview of the PNtalk architecture is presented in Figure 2. The architecture introduces a new meta level between the domain objects (i.e. PNtalk classes and PNtalk objects) and Smalltalk. Objects belonging to the metalevel are implemented by classes of Smalltalk. The metalevel comprises metaobjects which control PNtalk classes and PNtalk objects. A metaobject of the first kind describes *the structure* of a PNtalk class and it also defines the ways of manipulation with the PNtalk class. Metaobjects of the second kind describe *the computational behavior* of PNtalk objects (instances of PNtalk classes). Each such metaobject consists of a set of net instances (copies) and it also offers the means for structural modifications of the corresponding PNtalk objects. Thus both the PNtalk classes and the PNtalk objects are open to the dynamic changes.

The PNtalk metalevel architecture is based on objects, i.e. domain object does not really exist in the system but it is substituted (implemented) by its metaobject. This metaobjects is *actora*, thus it implements PNtalk concurrency model and serializes all accesses and requirements to the PNtalk object. A more detailed view on the PNtalk system and the PNtalk project can be found in [8, 9, 19, 12].

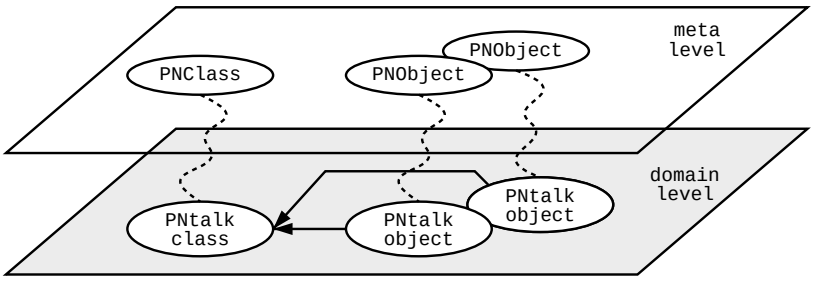

Figure 2. The PNtalk metalevel architecture – basic overview.

4.1 Interoperability

In the section 3.1 we have discussed a problem of interoperability between PNtalk objects and Smalltalk objects. To achieve this feature the *proxyobject* has been used in the PNtalk system architecture. 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

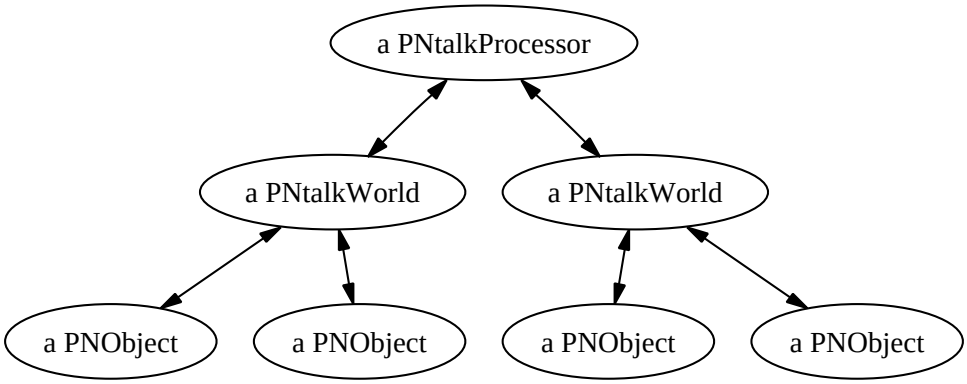

Figure 3. The architecture of the control metaobjects

handler (implemented in the proxy) decides how to react to the message. However, the PNtalk proxyobjects have to implement additional properties to save the prime metaobject protocol.

The architecture distinguishes several kinds of proxyobjects: a proxy for Smalltalk object, a proxy for PNtalk object (thus a domain level point of view) and a proxy for PNtalk metaobject (thus a metalevel point of view). Each object is referenced by means of the appropriate proxyobject (mostly the proxyobject is either for PNtalk object or for Smalltalk object) devolving incoming messages on the receiver in the suitable form.

The proxyobject does not define the computational behaviour of the receiver but, it ensures message passing in the right way (and at the requested level) and it conforms the response to sent messages in accordance with the requirements of the sender. Thus, the proxyobject adapts the computational behaviour of the receiver to the computational behaviour of the sender.

4.2 PNtalk metaobjects

Now we briefly introduce the basic metaobjects in the PNtalk system. We can see two parts here. Firstly, we have a description of a model structure and behavior – PNtalk classes. Secondly, we have a description of a simulation state – PNtalk objects. The PNtalk classe is substituted by its metaobjects (i.e. the instance of the Smalltalk class *PNClass*) including appropriate subcomponents (nets etc.). This part 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 (*PNClass*) or another name spaces. For the sake of simplicity we can suppose there is only the top-level namespace in the system.

The PNtalk object (i.e. the instance of the PNtalk class at the domain level) is substituted by its metaobjects (i.e. the instance of the Smalltalk class *PNObject* at the metalevel). They comprise appropriate subcomponents (processes etc.). Each PNtalk object is always placed into some world and each world is included into the PNtalk processor. That structure is shown in the Figure 3. We can see there following components:

- *PNObject* 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.

The Figure 4 presents overview of the basic metaobject composition (see *PNClass* and *PNObject*). The metaobject *PNClass* consists of *nets* (see *PNCompiledNet*). Each net represents a

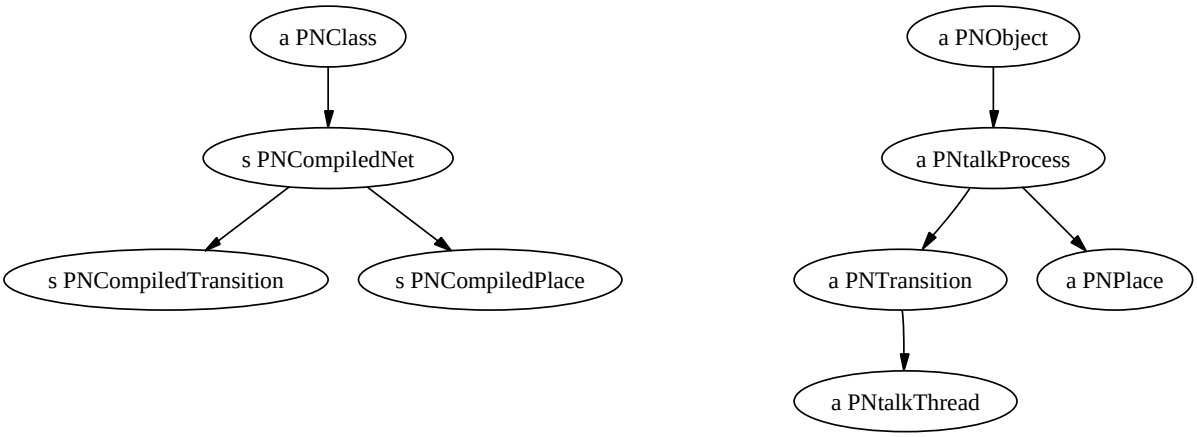

Figure 4. The composition of the metaobjects *PNClass* and *PNObject*.

translated method or object net of an appropriate PNtalk class. The net consists of places and transitions that are represented by metaobjects (see *PNCompiledTransition* and *PNCompiledPlaces*). When a new instance of the PNtalk class is being created then the metaobject *PNObject* is established. The metaobject *PNObject* consists of *procesess* (see *PNtalkProcess*). When some method is being invocated then a copy of the appropriate *net* 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 is represented by the thread (see *PNtalkThread*).

5 CASE STUDY – A MODEL OF BANK

As a demonstration of multisimulation capability of PNtalk, we present a simple case study featuring nested simulation – a model of bank. The specification of our model corresponds with the bank model introduced in [1, 21] (nested simulation was discused here in a context of the Simula language).

The model consists of two classes, and it is divided into five figures describing particular parts of the system. The sequence of parts that will be described corresponds to the process of model development in the PNtalk system.

5.1 Class BankModel

The class *BankModel* (see Figure 5) contains the object net describing arrival of customers (generated by transition *generator*) to the bank (with capacity of 100 customers). 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. This information is stored by the method *makeExperimentForPeriod:clerks:* (see Figure 6). This method initializes the model according to its parameters (a period number and a number of clerks). Furthermore, the transition experiment lets the simulation run for 600 time units (*self hold: 600*) and then it stops the simulation. We can experiment with this model (which will serve as a part of more sofisticated model later) by evaluating an expression like

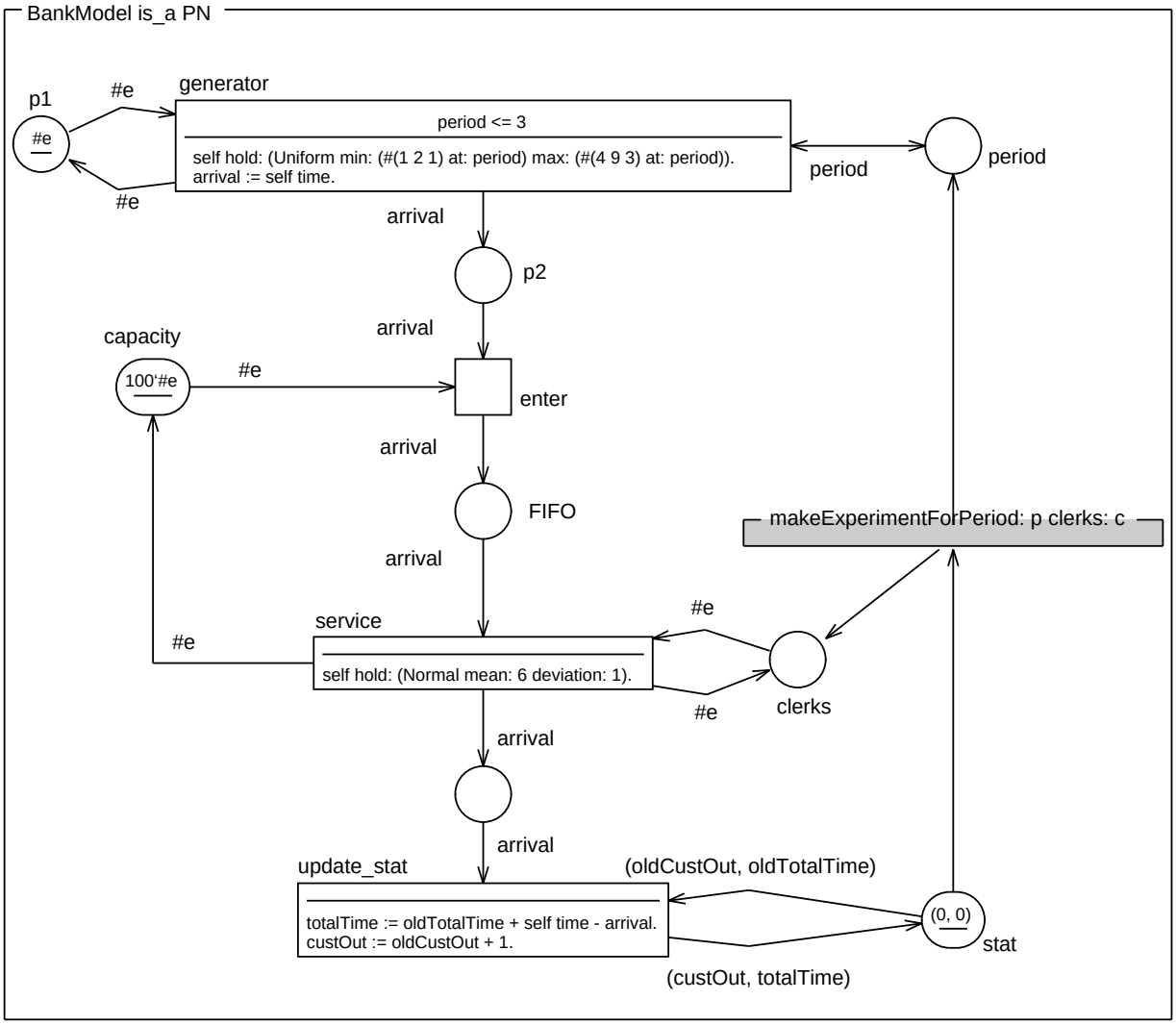

Figure 5. Object net of the class *BankModel*

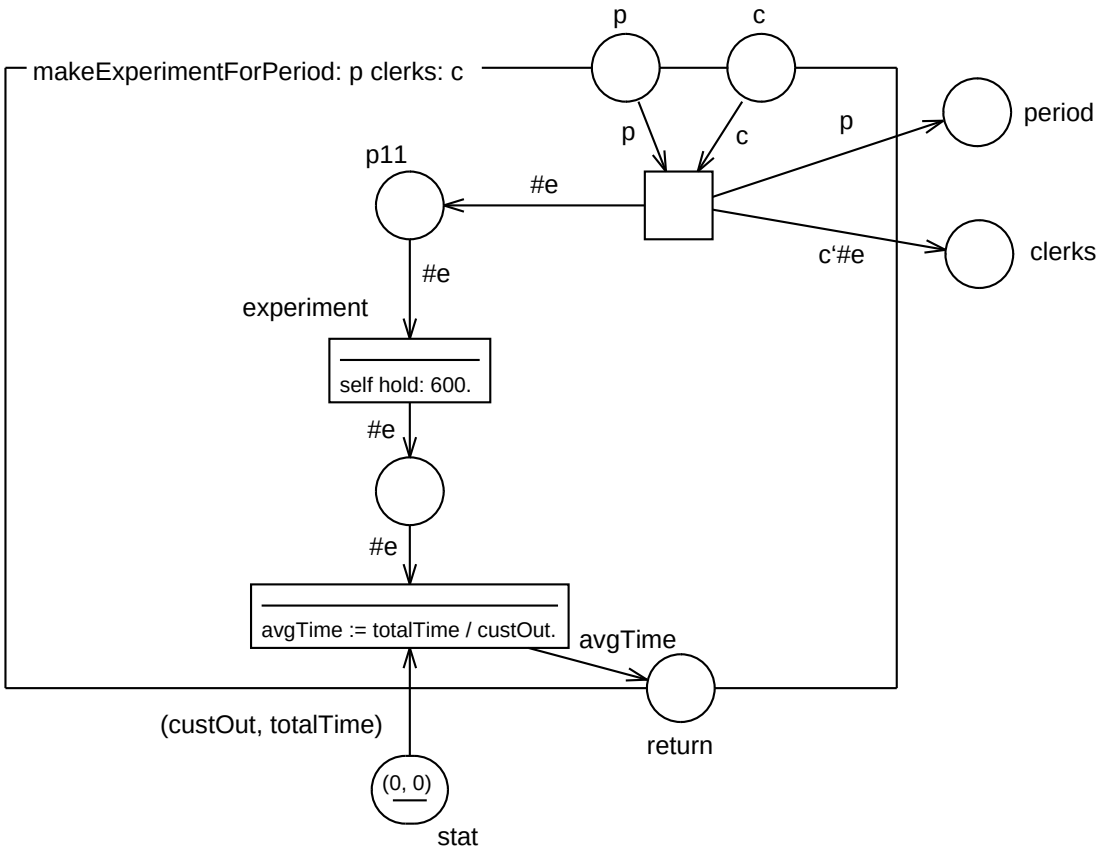

Figure 6. The method of the class *BankModel* controlling the experiment

BankModel new makeExperimentForPeriod: 1 clerks: 10

in Smalltalk workspace. It returns the average time the customers spent in the bank.

5.2 Class Bank

The class *BankModel* can serve as a basis for specialization by a more sophisticated model of bank. Moreover, we can show how an instance of *BankModel* can be simulated inside another simulation. The more sophisticated class *Bank* is derived (inherits) from the class *BankModel*. It adds second stage of service and a model of bank management making some decisions about assignment of clerks to first and second stage according to the results of nested simulation.

For better readability, the object net of the class *Bank* is divided into two figures. Figure 7 depicts the whole office. Figure 8 depicts the decision making part together with relocation of clerks assigned to the first and second stage of service. The inherited parts that are not modified are drawn by light-gray colour.

In the system, there are 10 clerks that are divided into two groups: for serving the first and second office. We will name these clerks as tellers (first office) and cashiers (second office). Their number is located in the places *tellers* and *cashiers*. Moving clerks between offices (tokens between places) is based on results of nested simulations for three situations during a whole simulation, and always at the beginning of a period of busy, idle, or very busy. The part of the object net that is doing decision making according with those results is shown on the Figure 8.

Now we explain how the simulation model works. An instance of class *Bank* represents the main simulation domain. As soon as the object is created we can distinguish two processes inside its object net: the simulation of bank offices and the decision part based on the nested

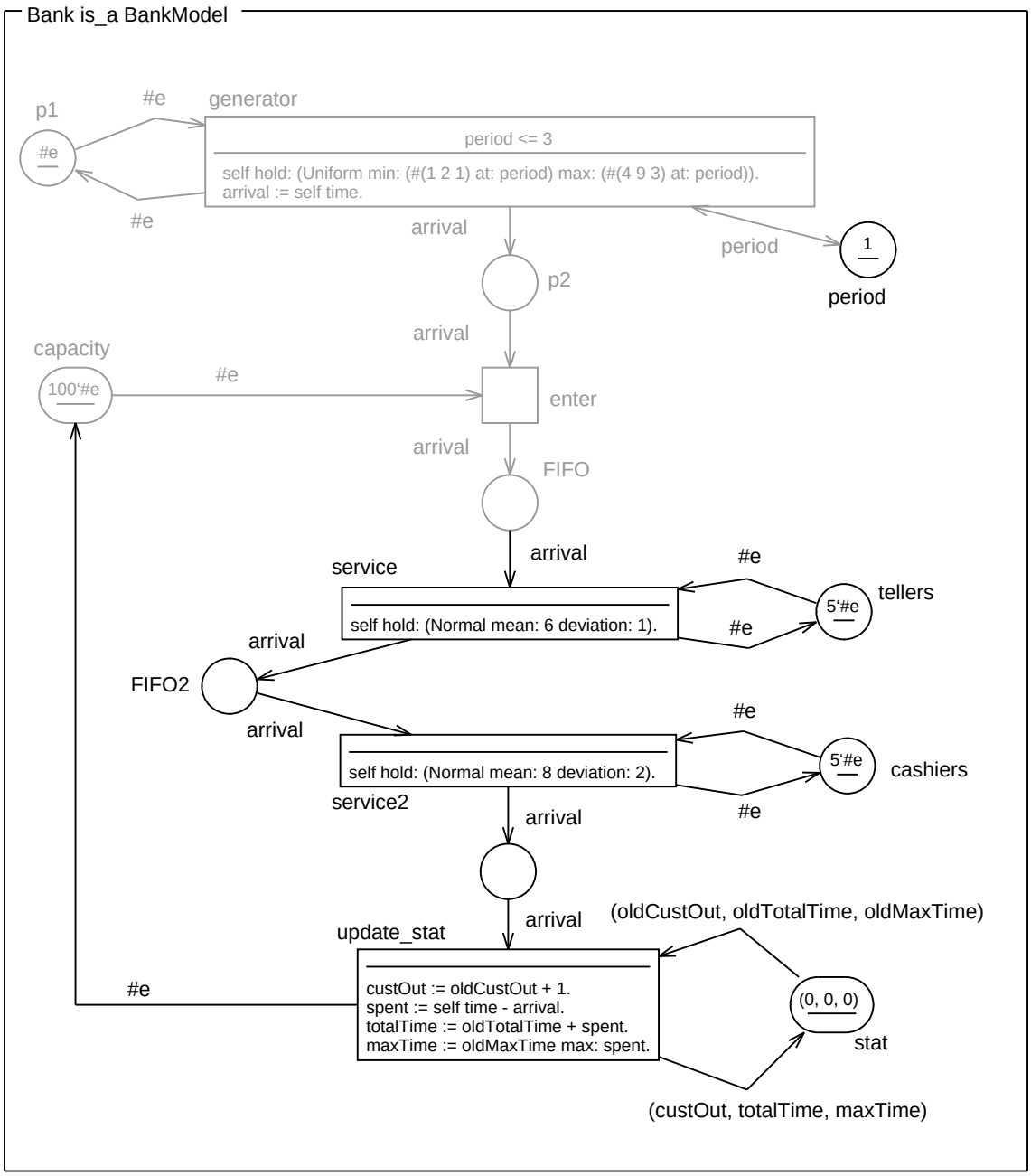

Figure 7. Object net of the class *Bank* – first part

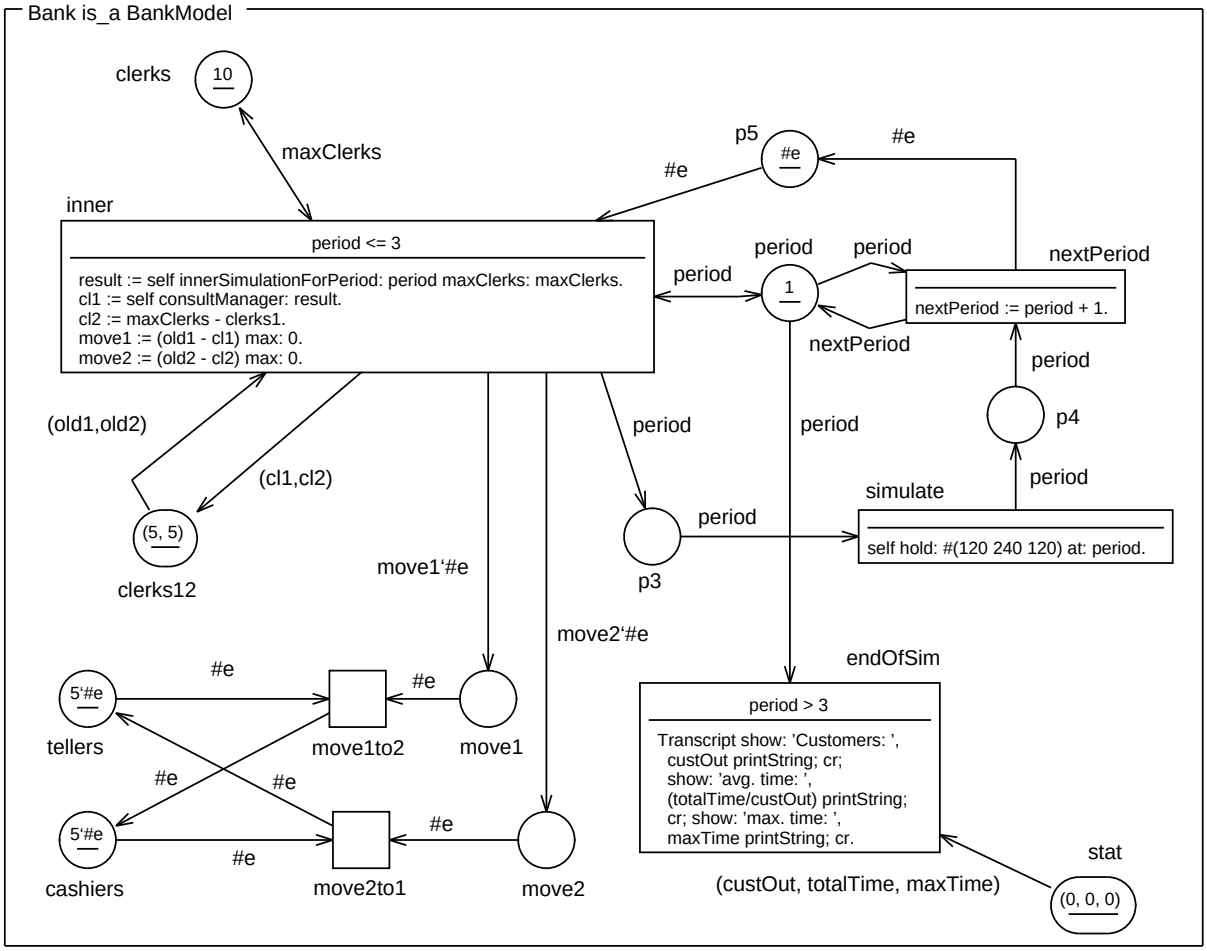

Figure 8. Object net of the class *Bank* – second part

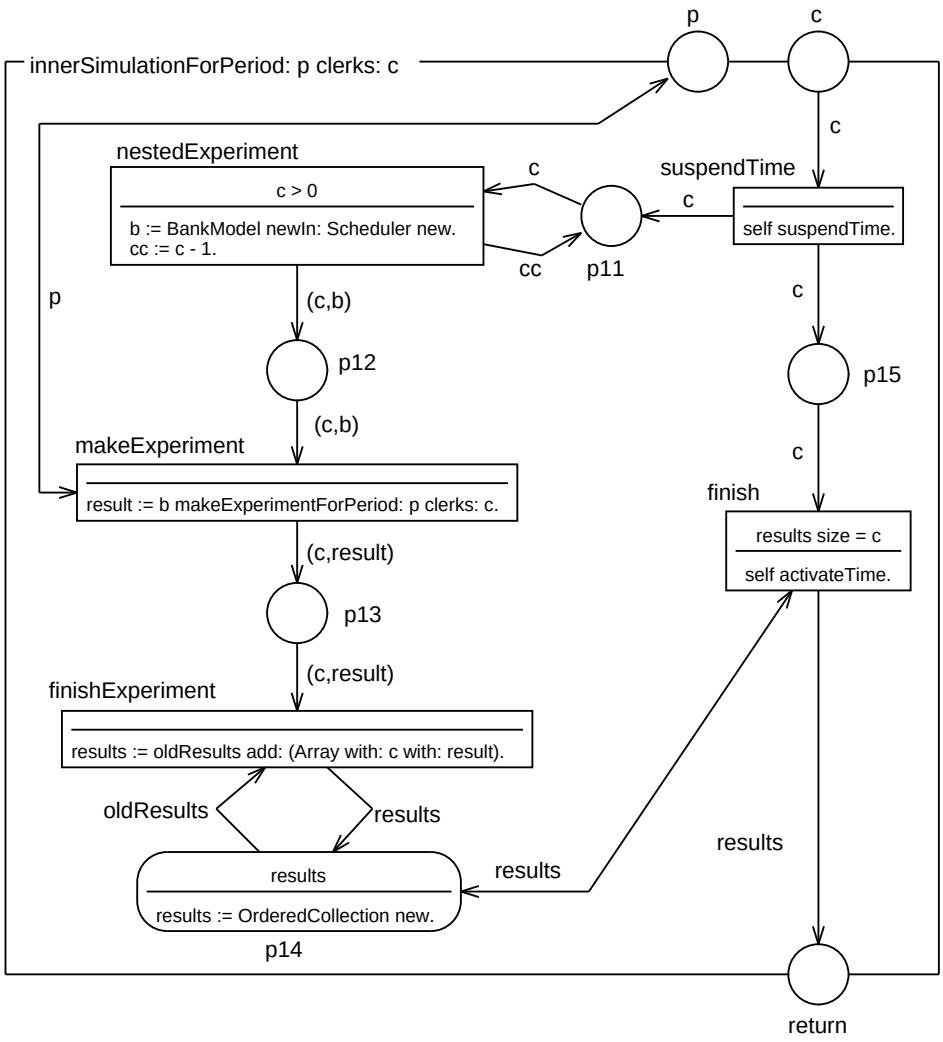

Figure 9. The method of class *Bank* controlling the nested simulation

simulation of simpler model represented by the class *BankModel*. The decision making part turns on the nested simulation for a period equal to a number 1 (it represents the busy mode). After finishing the nested simulation, the process makes a decision about how many tellers and cashiers will be needed in accordance with nested simulation's results (transition inner). This decision is not made automatically, but it is left up to the user. Recommended relocation of clerks is put into the places *move1* and *move2*. Now the model is simulated for the time which is specific for a given period (transition simulate). The transitions *move2to1* and *move1to2* realize the relocation. That situation is repeated three times (for period one, two and three) and then the simulation ends. Cycle $<$*p5, inner, p3, simulate, p4, nextPeriod, p5*$>$ ensures sequential processing of this part of the model.

5.3 Nested simulation

As we have said above the class *BankModel* provides the nested simulation which is controlled by the class *Bank*. Recall that nested simulation is started with two parameters: for which period the simulation will be held (the parameter p) and how many clerks are available (the parameter c). The Figure 9 depicts the *Bank* method providing start, controlling, and data collection of nested simulations. It starts the nested simulations step-by-step for c-times; the number of clerks is decremented in each next step. So the nested simulations prove all possible

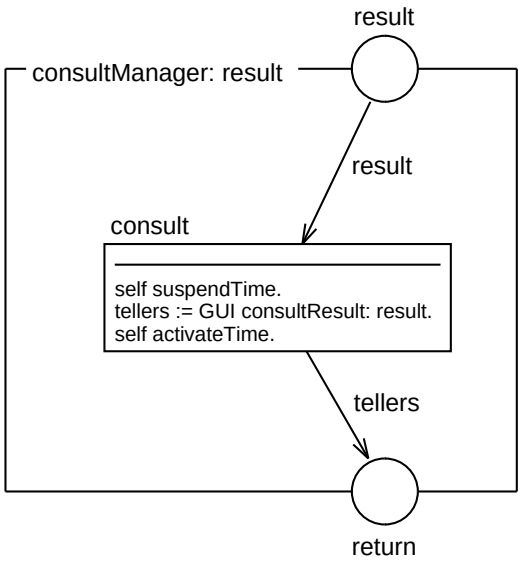

Figure 10. The method of class *Bank* ensuring the user's decision

situations with the number of clerks from c to 1. Since the nested simulation and decision about numbers of tellers and cashiers are planed as an atomic operation in the time we must guarantee the simulation-time will not change during an evolution of nested simulations. Therefore there are two special operations: *suspendTime* and *activateTime*. These operations are associated with the scheduler so that the suspension and the activation of time are valid for all objects attached to this scheduler. The time suspension means each event that may potentially occur in future (i.e. to finish a transition waiting for some time - see Figure 8, the transition *simulate*, the message *self hold:*) now cannot occur because the time does not change. All other events (which are planned to the current time) may occur.

Now we explain what happens during evolution of method *innerSimulationForPeriod:clerks:*. Firstly, the transition *suspendTime* suspends the time by means of an infdirect scheduler call *self suspendTime*. Secondly, the transition *nestedExperiment* generates *c* nested simulations specified by the class *BankModel*. The expression *BankModel newIn: Scheduler new* ensures that a new environment is created (scheduler) that is independent of other schedulers and that it has its own independent time axis. Then *a BankModel* is created and it is attached to the newly created scheduler. Finally, the reference to this object is returned (see the variable *b* in the Figure 9). Now we have to distinguish the basic environment (simulation of the bank) and the nested environments (nested simulations of *BankModel*). The basic simulation has suspended time. The transition *makeExperiment* starts the nested simulations.

As the time is suspended in the basic simulation environment the nested simulations are considered atomic from the view point of the time of the basic environment. As soon as all the nested simulations have been finished and the results have been collected (the transition *finishExperiment*) then the time is activated and the method returns the data (the transition *finish*). Evaluating an expression *Bank new* in Smalltalk workspace starts the simulation.

5.4 Simulation

The bank simulation is performed in three steps. In the first step, the inner simulation for first mode is being executed. Then the user does a choice of optimal number of tellers in the main simulation loop pursuant to inner simulation results (the screenshot of the bank model in PNtalk with results list and decision making by user is shown in the Figure 11). When the decision is made the main simulation loop runs for chosen number of tellers and cashiers according to

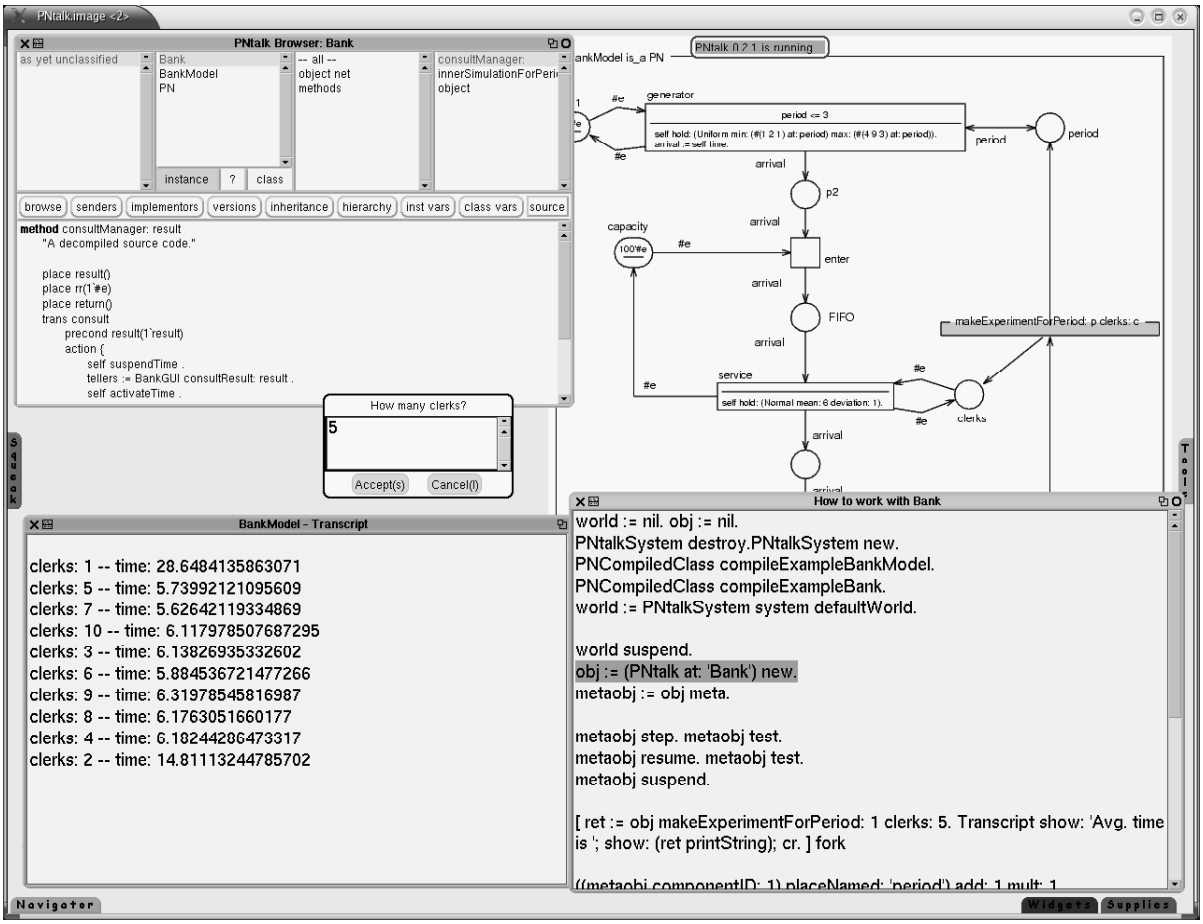

Figure 11. Screenshot of the case study simulation in the PNtalk system

generators and simulation time of first mode (*busy*). The next two steps are similar to first step with respecting values in the appropriate mode. The table 1 depicts simulation results of inner simulations for modes *busy*, *idle*, and *very busy* and for number of clerks form 1 to 10. The bold values represent chosen number of tellers for main simulation in the appropriate mode. The overall results of the main simulation (number of served customers, average time spent in bank by customers and the maximum time spent by customers) is shown in the table 2.

The presented example serves mainly for introduction the PNtalk modelling approach and nested simulation by using some of the reflective features of PNtalk. Of course, more interesting would be experimenting with automated optimalization (e.g. by replacing the user interface by some AI-entity in the model). Nevertheless, this is not a focus of this paper.

6 RELATED WORK

The PNtalk project is closely related with similar works. The idea of merging Petri nets and objects has been found in the first half of 1990s independently by several researchers. The probably best known issues have been introduced by Lakos [13], Sibertin-Blanc [20], Moldt [17], and Valk [24]. The Object oriented Petri nets and PNtalk language and system was developed in 1994 and published in [4, 6, 7].

The idea of object-oriented computational reflection (including structural and behavioral changes of objects at run time) was proposed in 1970s [5] but roots of this concept are much

	1	2	3	4	5	6	7	8	9	10
busy	28.65	14.81	6.14	6.18	**5.74**	5.88	5.63	6.18	6.32	6.12
idle	15.65	**5.62**	5.85	5.85	5.51	5.85	6.10	6.02	6.32	5.56
very busy	22.59	15.01	7.22	**5.89**	6.13	6.10	5.90	5.65	6.18	6.12

Table 1. The simulation results of nested simulations.

	number of customers	avg. time	max. time
main simulation loop	135	14.74	32.43

Table 2. The simulation results of the main simulation.

older (Lisp, Universal Turing machine). As the examples of recent activities the following projects can be cited – Kiczales [11] and Maes [15] introducing aspect-oriented programming, Actalk [3] introducing concurrency via reflection in Smalltalk, Apertos [25] and TUNES [22] are attempts to develop a highly flexible operating system using computational reflection. As to the reflection in modeling and simulation, the most important (from the PNtalk point of view) are Barros [2] and Uhrmacher [23] introducing reflectivity to DEVS [26] in order to allow for structural changes of models. These approaches are important especially for modeling and simulation of intelligent agents.

Our current attempts to incorporate reflection to the Petri nets also are not alone. Lakos [14] presents a reflective approach to Object Petri Nets implementation. He emphasizes the advantages of that approach in a clean, flexible and efficient implementation, and also in a possibility to investigate alternative scheduling schemes, interaction policies, etc.

7 CONCLUSION

There is a question what is the purpose of a system with a robust degree of the open approach moreover affiliated with the Object oriented Petri nets (OOPNs) formalism. The goal of PNtalk project is not only to make a tool intended for modelling and simulation but also to make tool allowing the developed model to be integrated to a real environment. Such a model can then serve as a part of the prototype or the target application. When we take in account the reflective features, we can use this system as a framework for interactive application development. The framework allows us to build models and prototypes, to combine different paradigms for model specification, to experiment with new paradigms, or to allow automatic evolution of models. In other words, the current trend in PNtalk development is a lite-form of an operating system (a lite-form in the sense of security features) primarily intended for a work with models and simulations.

One of possible application domain is artificial intelligence, especially the area of intelligent multi-agent systems. The process of the agent reasoning can be characterized by its inner structures changes. Moreover, the whole structure of a multi-agent system can dynamically change. The present goals of the PNtalk project are to check the promising features of open systems discussed above in practise and to check application of such system in the area of intelligent agent modelling and prototyping.

We have implemented a prototype featuring the presented architecture. We believe that the prototype will allow us to gain some experience needed for the development of the methodology and a further development of the PNtalk system itself.

REFERENCES

[1] The ASU Newsletter, vol. 27 no. 2, March 2002.

[2] F. J. Barros (1997). Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, v.7 n.4, 501-515.

[3] J. Briot (1999). Actalk : A Framework for Object Oriented Concurrent Programming - Design and Experience. In *Object-Based Parallel and Distributed Computing II - Proceedings of the 2nd France-Japan Workshop (OBPDC'97)*. Herms Science.

[4] M. Češka, V. Janoušek and T. Vojnar (1997). PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling. In *Computer Aided Systems Theory and Technology – EUROCAST'97*, volume 1333 of *LNCS*, 591–610, Spain. Springer-Verlag.

[5] A. Goldberg and D. Robson (1989). *Smalltalk 80: The Language*. Addison-Wesley.

[6] V. Janoušek (1995). PNtalk: Object Orientation in Petri nets. In *Proc. of European Simulation Multiconference ESM'95*, Prague, Czech Republic, 196-200.

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

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

[9] V. Janoušek and R. Kočí (2003). PNtalk: Concurrent Language with MOP. In *Proceedings of the CS&P'2003 Workshop*, Warsawa, PL, UW, 271-282, ISBN 83-88374-71-0.

[10] K. Jensen (1992). Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use, Volume 1, Springer-Verlag.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, Ch.Maeda, C. Lopes, J.M. Loingtier and J. Irwin (1997). Aspect-Oriented Programming, In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, Springer-Verlag, Berlin, Heidelberg, and New York, 220–242.

[12] R. Kočí (2004). Metody a nástroje pro implementaci otevřených simulačních systémů, Ph.D. thesis, Brno, CZ, p. 105.

[13] C. A. Lakos (1995). From Coloured Petri Nets to Object Petri Nets. In *Proceedings of the Application and Theory of Petri Nets 1995*, vol. 935, Springer-Verlag, Berlin, Germany, 278–297.

[14] Ch. Lakos (1996). Towards a Reflective Implementation of Object Petri Nets. In *Proceedings of TOOLS Pacific*, Melbourne, Australia, 129-140.

[15] P. Maes (1987): *Computational reflection.* Technical report, Artifical Inteligence Laboratory, Vrije Universiteit Brusel.

[16] H. Masuhara, S. Matsuoka, T. Watanabe and A. Yonezawa (1992). Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, SIGPLAN Notices, volume 27, number 10, ACM Press, New York, NY, 127–144.

[17] D. Moldt (1995). OOA and Petri Nets for System Specification. In *Application and Theory of Petri Nets; Lecture Notes in Computer Science*, 16th International Conference, Italy.

[18] R. Pawlak (1998). Metaobject Protocols For Distributed Programming. *Technical report*, Laboratoire CNAM-CEDRIC, Paris.

[19] PNtalk Project Home Page, http://www.fit.vutbr.cz/~janousek/pntalk

[20] C. Sibertin-Blanc (1994). Cooperative Nets. In *Proceedings of Application and Theory of Petri Nets*, vol. 815, Springer-Verlag, Berlin, Germany, 471–490.

[21] J. Sklenar (1997). Introduction to OOP in Simula. *http://staff.um.edu.mt/jskl1/talk.html*.

[22] The TUNES Project for a Free Reflective Computing System, http://tunes.org

[23] A. M. Uhrmacher (2001). Dynamic structures in modeling and simulation: a reflective approach. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, Volume 11, Issue 2, 206–232, ISSN:1049-3301.

[24] R. Valk (1998). Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, Vol. 1420, Springer-Verlag, Berlin, 1-25.

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

[26] B. Zeigler, T. Kim and H. Praehofer (2000). Theory of Modeling and Simulation. *Academic Press*, 2nd edition, 510 pages, ISBN: 0127784551.