PNtalk - An Open System for Prototyping and Simulation

Vladimír Janoušek
Radek Kočí
Department of Intelligent Systems
Brno University of Technology, Božetěchova 2, Brno 612 66, Czech Republic
Web: http://www.fit.vutbr.cz/~janousek
http://www.fit.vutbr.cz/~koci
E-mail: {janousek, koci}@fit.vutbr.cz

Abstract

The PNtalk system is a language and tool based on object-oriented Petri nets (OOPNs) that have been developed by our research group at Brno University of Technology. Its purpose is to support modelling and prototyping of concurrent and distributed software systems. 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 PNtalk system was originally designed as a transparent layer on top of a Smalltalk system. The Smalltalk classes and objects are transparently interchangeable with classes and objects described by Petri nets. Nevertheless, the original implementation had some limitations disallowing evolution towards distributed programming and advanced approaches to simulation. This is the reason why we decided to redesign the PNtalk system to employ reflective features of Smalltalk-80 more effectively. Our current prototype has been built up as an open system which can combine different modeling and simulation paradigms of describing models and can also simultaneously support several independent simulations.

This article is focused on describing the PNtalk architecture. A simple yet suggestive model will demonstrate how PNtalk can be used for modelling and simulation.

1. Introduction

For some years, members of our research group have been conducting experiments with some prototypes of a hybrid language for prototyping and simulation of parallel and distributed systems. We are particularly interested in exploiting a Petri net-based formalism for such purposes because of its ability to express parallelism and data flow suggestively. We consider Petri nets as something more than an abstract mathematical formalism - we believe that Petri nets could be used as a kind of a parallel programming language. PNtalk is an experimental language based on that idea. It merges Smalltalk and Petri nets. The formalism behind PNtalk is called Object-oriented Petri nets (OOPN).

The PNtalk system was originally designed as a transparent layer on top of a Smalltalk system. The Smalltalk classes and objects are transparently interchangeable with classes and objects described by Petri nets. Up to now we considered PNtalk to be used mainly for prototyping. This means that all models run in real time and communicate with real-world surroundings. In fact, we have also experimented with simulations a bit, but our simulation models were very simple. For such purposes, a global time variable was sufficient. No advanced concepts were considered. Nevertheless, we are going to consider them in the current PNtalk design. A well-known example of such advanced concepts is nested simulation [10,11,12,13]. We are going to employ this concept as a part of a case study, which will allow us to show and discuss some important features of the current PNtalk design.

This paper is organized as follows. We first present the main ideas behind OOPN and PNtalk language and system. Then we describe meta-level architecture of PNtalk. Finally, we present a simple yet suggestive model, which demonstrates how the PNtalk can be used for modelling and simulation.

2. 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 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) [5]. 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) [2,3]. 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.

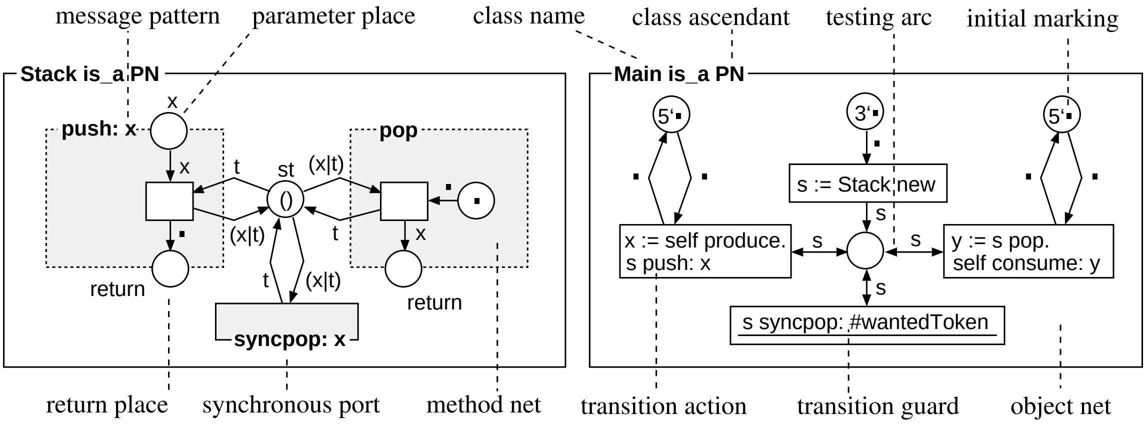

Figure 1: An OOPN example (*Main*'s methods *produce* and *consume* are not shown).

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). Let us describe the transition semantics. A message that is sent to a primitive object is evaluated atomically, 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. Then, the output part of the transition can be performed. In the case of the transition guard, it is possible to send messages only to non-primitive objects with appropriate synchronous ports.

Inheritance

Classes are defined incrementally using inheritance. A general superclass of all classes defined by Petri nets is class PN that contains an empty object net and an empty set of method nets. Nevertheless, it defines common behavior of all active objects in the PNtalk system as well as access to some metainformation (e.g. if it is primitive object). Class inheritance is defined by the inheritance of object nets (inherited transitions and places identified by their names can be redefined and new places or transitions can be added by a subclass; places can redefine initial marking, transitions can redefine their preconditions, postconditions, guards, and actions), together with sets of method nets (the implementation of inherited methods can be replaced and new methods can be added by a subclass).

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 [2]. The client-server interaction mechanism in OOPN allows for such interoperability.

If the PNtalk is implemented in Smalltalk (i.e. as a part of Smalltalk system), all PNtalk objects can be implemented by Smalltalk objects, and the message sending mechanism of PNtalk can be implemented directly by the message sending mechanism of Smalltalk. This means that PNtalk objects are 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).

Time

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. This is one of the important features of the current PNtalk design.

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.

3. The PNtalk System Architecture

The whole PNtalk architecture is built up as a meta-level system which means it distinguishes two kinds of behaviour: the so-called domain behaviour and the computational behaviour of objects. Obviously the computational behaviour is latent (encapsulated) in a language environ-

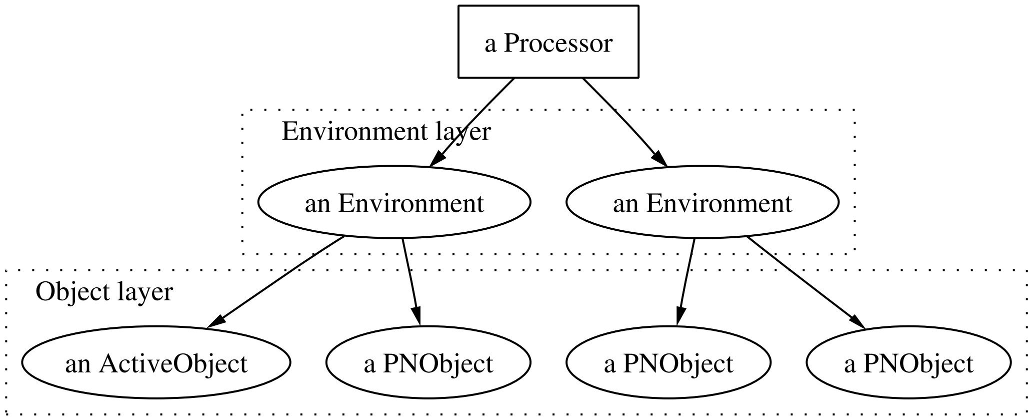

Figure 2: The System Architecture

ment such as C++, Simula or Java. However, in many cases there may be a requirement for a means enabling a change of computational behaviour of an object. An example of that property may be a connection of objects that are described by different paradigms. Now we are focused on connectivity between objects described by Petri nets and objects of Smalltalk.

Objects of these two paradigms have different computational behaviour whereas this fact should not be visible at domain level. Said in another way, the object (sender) sends a message to another object (receiver) in a uniform way and it does not need to know what kind of object the receiver is. This is the basic purpose for a division of the behaviour into two parts: the computational and domain behaviour. Therefore in further text we will be distinguishing between domain-object and meta-object. The meta-object implements the computational behaviour of its associated domain-object. This means that the domain-object behaviour is controled by the meta-object. This principle is a basic idea of the new simulation system architecture that has to satisfy some features. Firstly, the different sets of objects can have different computational behaviours. Thus the architecture has to enable a communication between these sets of objects. This property is called interoperability. Secondly, a set of objects represents a simulation that is in particular characterized by its own time axis. Several independent simulations can run in the system.

We can trace up some hierarchical relationships between components of the system. There are objects making up a model, meta-objects controlling those objects and the simulation environments controlling meta-objects. Thus we can speak about a multi-level architecture. The conceptual structure of the system is shown in the Figure 2. In next text, we describe particular layers and their influence on the simulation system.

The object layer

It represents all non-primitive domain-objects that are being created during an evolution of the model. In the system, the domain-objects are controlled (implemented, interpreted) by the meta-objects. Smalltalk's objects do not have associated meta-objects. To ensure uniform communication between arbitrary domain-objects (for example between a Smalltalk object and an active object controlled by its meta-object) all the domain-objects are always represented by the proxy-objects.

The proxy-object

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.

The proxy-object in the PNtalk architecture has a few significant properties: to ensure communication between objects having different computational behaviour, to ensure uniform communications between objects and to enable message passing to distributed objects without the need of message sending modifications.

An object (potential sender of a message) always refers to another object (potential receiver) by means of a proxy-object devolving incoming messages on the receiver in the suitable form. In spite of the domain-level point of view, the actual sender or receiver is either the domain-object (when it is a regular Smalltalk object), or the meta-object (when the receiver is nested in the simulation architecture).

The proxy-object does not define the computational behaviour of the receiver but, it ensures message receiving by the receiver in the right way and it conforms the response to sent messages in accordance with the requirements of the sender. Thus, the proxy-object adapts the computational behaviour of the receiver to the computational behaviour of the sender.

The environment layer

The environment defines a life-space of a set of objects. It controls cooperation of objects in the environments. Furthermore it offers a notion of time. Thus, it allows grouping the objects to the relative independent units. Every environment has its own scheduler. In many cases it is obvious to interchange the notions environment and scheduler mutually. Objects can migrate between environments and can communicate with objects of other environments. We can view the environment as a means determining the simulation or a part of the simulation.

Processor

At the top of the architecture there is an object named Processor. It ensures the cooperation of all environments in a Smalltalk image. We can say it is the controlling center of the PNtalk system.

4. Case Study – The Model of Bank

In this chapter, we describe a simple model of a bank. The model consists of two classes, and it is divided into five figures describing particular parts of the developing system. The sequence of parts that will be described corresponds to model development in the PNtalk environment. The specification of our model corresponds with the bank model represented in [12,13].

Class BankModel

The class *BankModel* (see Figure 3) 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 4). 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

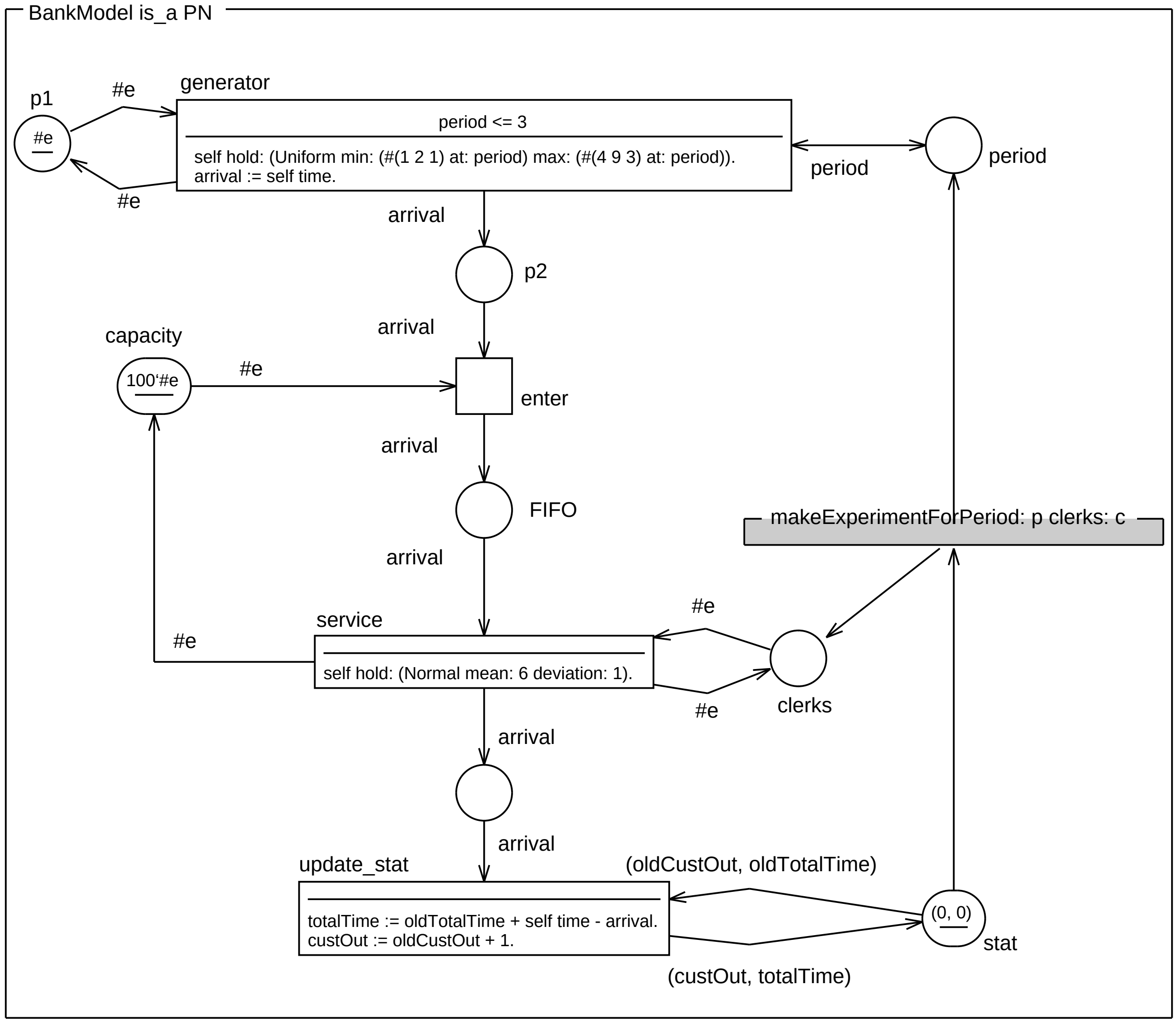

Figure 3: Object net of the class *BankModel*

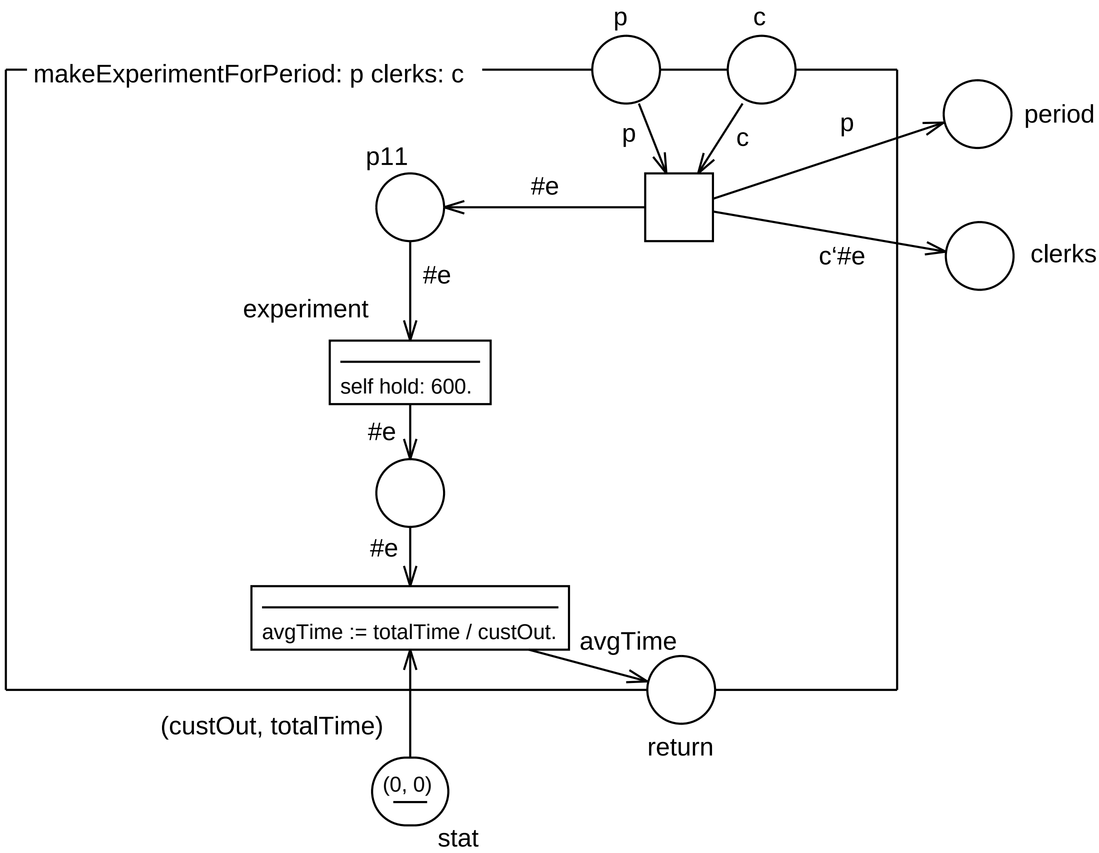

Figure 4: The method of the class *BankModel* controlling the experiment

the simulation. We can experiment with this model by evaluating an expression like

BankModel new makeExperimentForPeriod: 1 clerks: 10

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

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 5 depicts the whole office. Figure 6 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

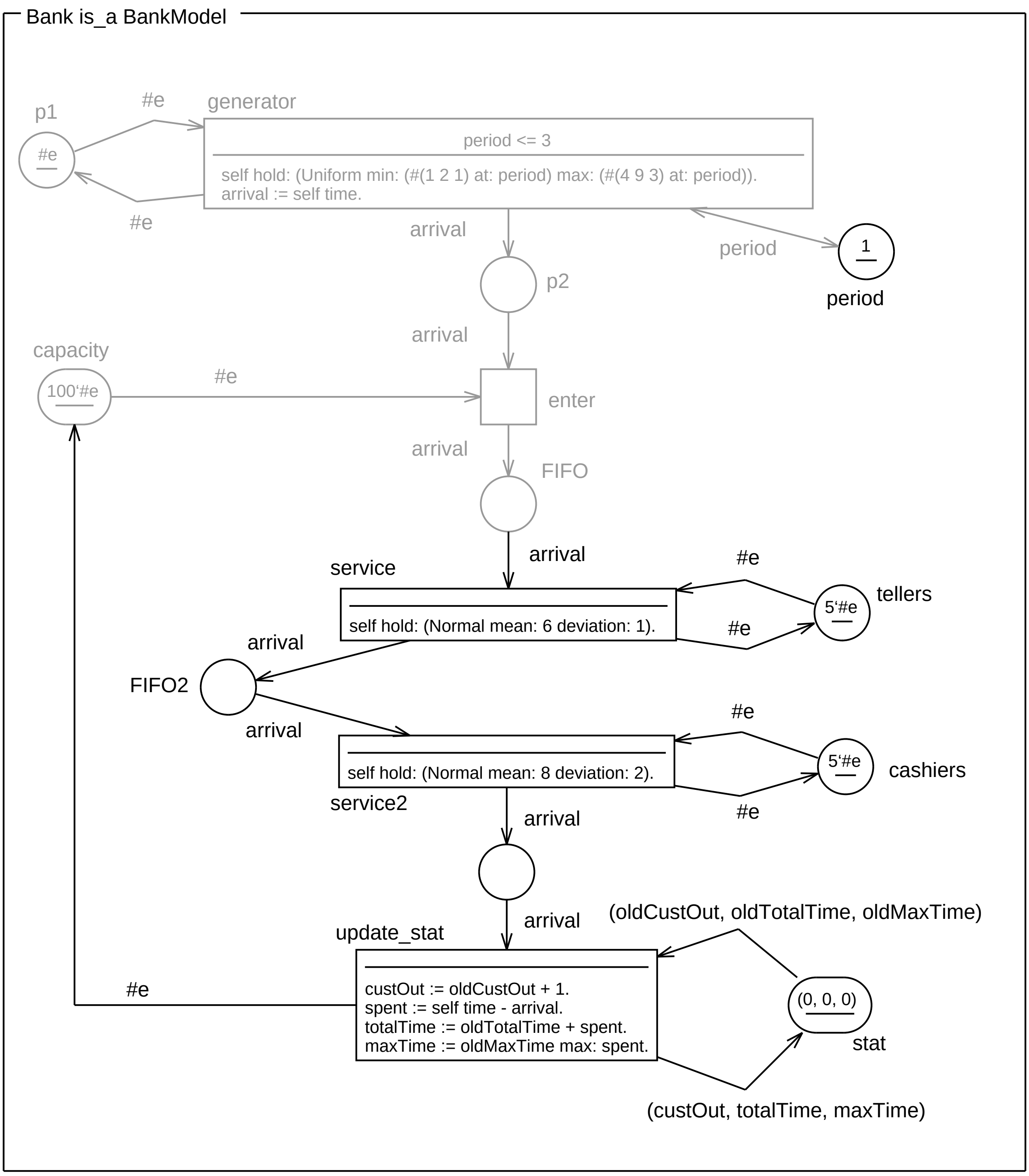

Figure 5: Object net of the class *Bank* – first part

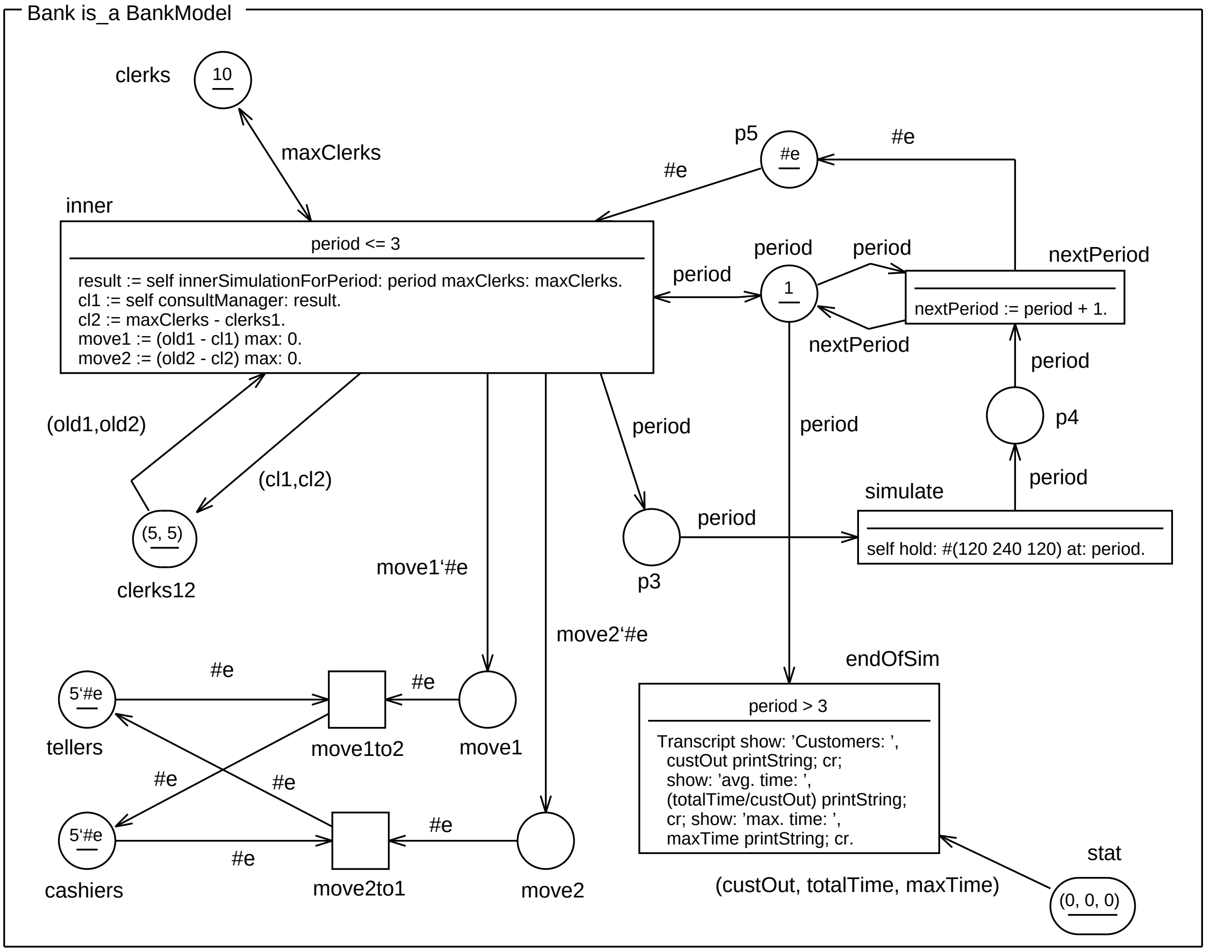

Figure 6: Object net of the class *Bank* – second part

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 6.

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 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.

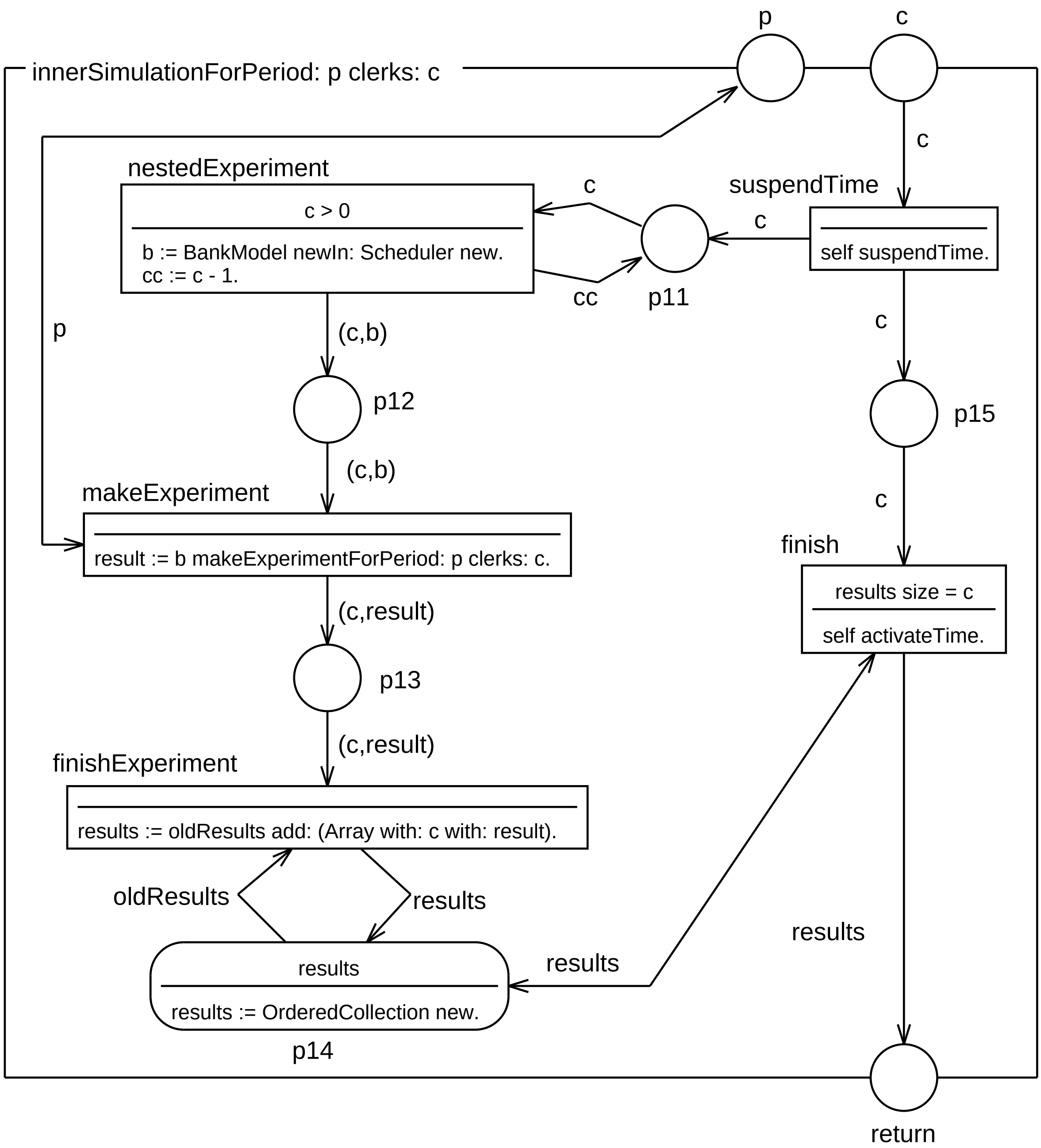

Figure 7: The method of class *Bank* controlling the nested simulation

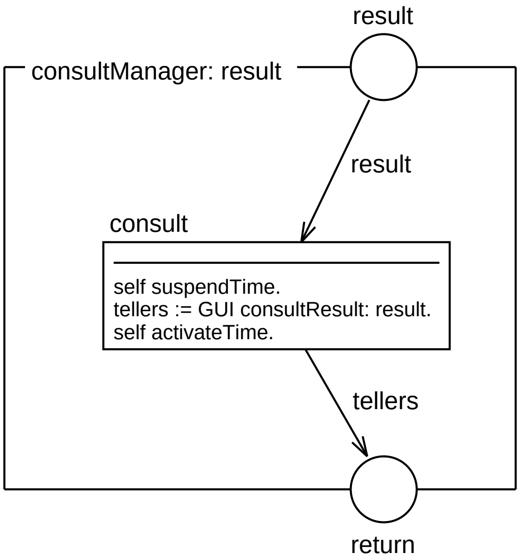

Figure 8: The method of class *Bank* ensuring the user's decision

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 7 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 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 6, the transition *simulate*, the message *self hold:*) now cannot occur because the time does not change. All other events (that do not use time) can 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 7). 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.

Conclusion

The PNtalk system, its architecture, and the case study of its use have been presented in this paper. PNtalk is being developed as the tool for prototyping and simulation. This paper was focused on simulation. The architecture of the current PNtalk system implementation allows for multiple simulations and reflection [10, 11]. These features were shown on the well-known example of a bank model with nested simulations.

At this time we have implemented the prototype of the PNtalk system supporting all the features described above. In the near future we plan to employ the open architecture of PNtalk as a basis for further development towards distributed and heterogeneous simulations.

Acknowledgement

This work was supported by the Grant Agency of the Czech Republic under the contract GACR 102/00/1017 “Modelling, Verifying, and Prototyping Distributed Applications Using Petri Nets”.

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. PNtalk: Object Orientation in Petri Nets. In Proceedings of European Simulation Multicoference ESM'95, pages 196(200, Prague, Czech Republic, 1995. Czech Technical University.

[3] V. Janoušek. Modelling Objects by Petri Nets. PhD thesis, Department of Computer Science and Engineering, Technical University of Brno, Czech Republic, 1998. (In Czech).

[4] M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. Modelling with Generalized Stochastic Petri Nets, Wiley Series in Parallel Computing, 1995.

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

[6] M. Češka and V. Janoušek. Object Orientation in Petri Nets. In Object Oriented Modelling and Simulation, Proceedings of the 22nd Conference of the ASU, pages 69–80, Clermont-Ferrand, France, September 1996. University Blaise Pascal.

[7] V. Janoušek and T. Vojnar. The PNtalk Project, 2000.
http://www.fee.vutbr.cz/˜janousek/pntalk/pntalk.html and
http://www.daimi.au.dk/petrinet/tools.db/pntalk.html

[8] C.A. Petri. Kommunikation mit Automaten. PhD thesis, Institut fűr Instrumentelle Mathematik, University Bonn, Germany, 1962. Available as Schriften des IIM Nr. 2. (In German).

[9] J. Peterson. Petri Net Theory and the Modelling of Systems. Prentice Hall, Engelwood Cliffs, New Jersey, 1981.

[10] E. Kindler et al. A Contribution to Reflective Simulation of Systems of Queuing Management. In: Proceedings of MOSIS 2001. MARQ. 2001.

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

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

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