PNtalk – a Computerized Tool for Object Oriented Petri Nets Modelling*

Milan Češka, Vladimír Janoušek, and Tomáš Vojnar

Department of Computer Science and Engineering, Technical University of Brno
Božetěchova 2, CZ–612 66 Brno, Czech Republic
e-mail: {ceska,janousek,vojnar}@dcse.fee.vutbr.cz

Abstract: *The paper deals with a new formalism and a tool for concurrent systems modelling and prototyping. This formalism, Object Oriented Petri Net (OOPN), combines the descriptive power of Petri nets with the well known advantages of object oriented modelling techniques. In the paper there are presented the structure of system specifications by means of OOPNs, principles of their dynamic behaviour as well as some basic features of a computerized tool which supports practical use of OOPNs. This tool works with a language called PNtalk which is based on the OOPNs and the language Smalltalk.*

Key words: Petri nets, object orientation, visual tools, simulation modelling and prototyping.

1 Introduction

Nowadays there is clearly visible a movement in the field of new system design methodologies and computerized tools for systems analysis, prototyping, and implementation. These tools and methodologies often reflect paradigms and theoretical concepts from the System and Computer Science. Many examples of such a reflection can be recognized in current research projects as well as in some available tools for system engineering. They are connected to automata, different kinds of logic, grammars, neural nets etc., or to completely new concepts. The developed theories of the corresponding theoretical models can then extend and support applications of the resulting tools in the areas of system verification or optimization.

1.1 Petri Nets and Objects

During recent years several authors have been trying to combine the Petri nets formalism with object oriented modelling techniques in order to develop a formal environment for system specification and design which would preserve advantages of both the paradigms, i.e. natural ways of modelling parallelism, formal semantics and analysis methods provided by Petri nets and concepts of abstraction, encapsulation, reusability or inheritance provided by object orientation. Resulting formalisms can be characterized by the degree to which they support appearance of objects inside nets and nets inside objects.

*A slightly modified version of this article was presented within the EUROCAST'97 conference, Las Palmas de Gran Canaria, Canary Islands, Spain, 1997 and will be published in the Springer-Verlag Lecture Notes in Computer Science series.

For example the Thorn project [8] has lead to a formalism which allows us to bind C++ objects to tokens, but there is no possibility of having nets inside objects here. The CodeSign project [14] deals with Petri nets in objects, but objects of classes described by Petri nets can be created only statically, i.e. before the start of execution. The LOOPN++ formalism [3] allows tokens to be bound both to C++ objects as well as to dynamically created objects described by particular Petri nets. However, there is no concept of methods described by Petri nets here.

In this article there is presented another approach to combining Petri nets and objects, which is somewhat related to the one described in [4]. This is because it also uses the RPC mechanism for objects communication. Apart from that, the developed formal model, Object Oriented Petri Net (OOPN), is based on the concept of net invocation [1]. The OOPN formalism is used as a base of the tool and language PNtalk, which are intended for executable formal specifications in the Smalltalk environment. It works with objects inside nets and also with nets inside objects. These nets describe independent objects activities as well as methods as reactions to messages sent from outside of objects.

When describing the OOPN formalism, we assume the reader to be familiar with basic concepts and notions of object oriented systems and Petri net models. The reader should also bear in mind that the description presented here is mostly informal.

1.2 Fundamental Principles of OOPNs and PNtalk

The proposed OOPN formalism is characterized by a Smalltalk-based 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 which offer reentrant services to other objects. Services provided by objects as well as independent activities of objects are described by Petri nets – services by *method nets*, object activities by *object nets*. Tokens in nets represent references to objects.

An OOPN consists of Petri nets organized in classes. Every class consists of an object net describing the internal activity of objects of this class and a set of method nets describing how these objects respond to messages. All method nets share access to the object net (places of object net are accessible by transitions of method nets). Each method net is reentrant and has parameter places and a return place. Class inheritance is defined by the inheritance of object nets, together with sets of method nets.

Each object is an instance of some class and consists of an instance of its object net and currently running instances of its method nets. When an object receives a message, a new instance of the corresponding method net is created, parameters are put into the parameter places and the instance of the method net is executed concurrently with all other net instances until the return place receives a token. Then the value of the token in the return place is passed to the message sender, and the instance of the method net is deleted. Message sendings and object creations are specified as actions attached to transitions. It is a polymorphic modification of the above mentioned invocation – a message receiver itself decides which method has to be invoked in order to respond to a message (according to the receiver's class).

2 OOPN Models Structure and the PNtalk Language

In this section there is given an informal specification of the structure of the object oriented Petri nets. A formal specification can be found e.g. in [12, 13]. To give the reader a general information before looking at details we can say that an OOPN consists of classes, every class consists of nets and those are built of places and transitions interconnected by arcs in a way common in the area of high-level Petri nets. Places, transitions, and arcs have their inscriptions. Every place has an initial marking and an initial action. Transitions have guards and actions. Finally, arcs are joint with arc expressions. All features of OOPNs, including net inscriptions, can be described in a graphical as well as textual way by means of the language PNtalk.

2.1 PNtalk Inscriptions

The PNtalk language is based upon the purely object-oriented language Smalltalk. Because no types of variables are introduced, no declarations are necessary, and therefore net inscriptions can be rather simple and easily readable.

Terms of the PNtalk language can be listed as follows:

1. *Literals.* They identify primitive objects of PNtalk:

   (a) *Numbers.* Numbers are objects which represent numerical values and respond to messages requiring results of mathematical operations. Numerical literal is a sequence of digits, possibly preceded by a sign and/or followed by a point and another sequence of digits. It is also allowed to use exponents, which begin with the character `e`. For example `5`, `-23`, `0.01`, `1.5e-3`.

   (b) *Characters.* Characters are objects which represent particular symbols of ASCII code. Their literals begin with the dollar sign. Some examples are: `$a`, `$B`, `$7`, `$$`.

   (c) *Strings.* Strings are objects representing sequences of characters. They respond to messages of character access and string comparison. String literal is a sequence of characters surrounded by apostrophes. Apostrophes inside strings must be doubled: `'abcd'`, `'can''t'`.

   (d) *Symbols.* Symbols are objects which are typically used as names. They are represented by sequences of characters preceded by the character `#`, e.g. `#e`, `#abc`, `#a1`. It is guaranteed that two symbols named by the same character sequences refer to the same object. This allows more effective comparing than in the case of strings.

   (e) *Boolean literals.* They are referred to by the reserved identifiers `true` and `false` and are used as results of all boolean expressions.

   (f) *Undefined object.* It is represented by the reserved identifier `nil`, and e.g. all variables which have not been assigned any value yet point to this object.

2. *Variables.* They can represent different objects as the OOPN evolution goes on. They are written as identifiers, i.e. sequences of characters and digits beginning with low case characters, e.g. `x`, `y`, `anInteger`.

   Some identifiers are reserved as literals (`true`, `false`, `nil`) and some of them can be used as names of classes only. The latter ones begin with capital letters, e.g. `PN`, `Object`, `C1`. Two identifiers are also reserved for the so called pseudovariables (see below).

3. *Pseudovariables.* These terms change their value according to the context in which they are used but their value cannot be changed progammably. There are two of them – `self` and `super`.

Expressions in PNtalk have the form of terms, *message sendings* or *variable assignments*. A message sending has the form $\langle receiver \rangle \langle message \rangle$. A message consists of a selector and possible arguments. There are three types of messages here:

1. *Unary messages.* Selectors of these messages are identifiers which do not contain a colon. These messages do not have any arguments, e.g. `5 factorial`, `C1 new`.

2. *Binary messages.* They have one argument and correspond to the usage of common operators like `+`, `-` etc. An example can be `2 + x`.

3. *Key word messages.* Their selectors contain at least one colon, and therefore at least one argument must be used, e.g. `x between: 1 and: 10`, `anArray at: anIndex`.

Variable assignments can appear in transition actions but not in transition guards (will be introduced later on). They have the form $\langle variable \rangle := \langle expression \rangle$, e.g. `y := x + 1`.

There can also be used *sequences of expressions* here. The dot sign plays the role of a delimiter, e.g. "`y := x + 1.  z := y factorial`". These sequences correspond to conjunctions in transition guards and to regular sequences of commands in transition actions.

Arc expressions in PNtalk represent multisets and can be syntactically described in the following way: $n_1\text{‘}c_1, n_2\text{‘}c_2, \ldots, n_m\text{‘}c_m$. Here n_i must be a term and c_i is a term or a list for all $i \in 1, 2, \ldots, m$. If n_i is equal to 1, it can be omitted. *Lists* have the form $(h_1, h_2, \ldots, h_m \mid t)$ where $h_1, h_2, \ldots, h_m$ represent elements at the beginning of the list and t corresponds to the rest of the list. The construction representing the rest of a list can be left out and thus we obtain a notion of an m-tuple written in the form $(h_1, h_2, \ldots, h_m)$. There are some examples of arc expressions here: `#e`, `2‘#e`, `3‘1`, `(x, 1 | y)`.

2.2 Places, Transitions, and Arcs

A PNtalk *place*, graphically represented as $\bigcirc$, is determined by its name, initial marking, initial action, and, of course, by the net in which it appears. An *initial marking* is specified syntactically as an arc expression. If there appear some variables in the initial marking, their values must be set by the *initial action.* Initial actions have the same form as transition actions. Places are capable of holding *tokens.* Tokens represent references to objects, which can be either trivial or defined by OOPNs.

A *transition* in PNtalk, graphically drawn as $\square$, is determined by its name, guard, action, and the net in which it is situated. A *guard* of a transition is a conjunction of expressions which must be atomically evaluable, and which are separated by dots. Unlike in guards, the particular expressions in a sequence which creates a transition *action* need not to be atomically evaluable.

Places and transitions of OOPNs are interconnected by *arcs.* Every arc has an associated *arc expression* whose syntax we have already described. Arcs are considered to belong to transitions. From this point of view, there are three different kinds of arcs here:

1. *Input arcs,* $\rightarrow\square$, which remove tokens from input places of transitions.

2. *Output arcs,* $\leftarrow\square$, which add tokens into output places of transitions.

3. *Dialog arcs,* $\leftrightarrow\square$, which only test whether there are appropriate tokens in input places but do not remove them. They have special meaning for non-atomic transitions (to be explained later on).

The above described notion of places, transitions and arcs is demonstrated in Fig. 1.

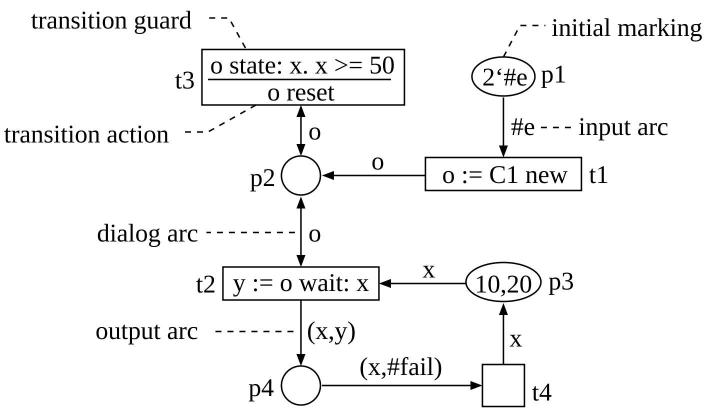

Figure 1: A demonstration of the notion of OOPN places, transitions, and arcs

2.3 Method nets and object nets

Places and transitions with arcs constitute *nets*. PNtalk distinguishes two kinds of nets – object nets and method nets. These nets are used for specification of classes of objects, which can communicate through message sending.

Object nets represent attributes of objects and their implicit activities, i.e. activities which are not straight responses to some messages sent from outside of the objects. In this way, so called *active objects* can be specified. An example of such a net is the one shown in Fig. 1.

Method nets specify how particular objects respond to incoming messages. They consist of interconnected places and transitions like the object nets. But what is more, a message pattern is associated with every method. Matching messages can then be used for dynamic creating of method net instances. There exists a special, possibly empty, subset of the set of all method net places which are called *parameter* places. Parameter places are used for argument passing during method invocations. Their names must be the same as the names of formal parameters in the appropriate message pattern. Further, every method has one special place, called *return* place, which is used for obtaining results from the method.

Method and object nets can be interconnected in such a way that there can exist arcs from object net places to method net transitions and vice versa. Thus methods can test the state of the object over which they work and possibly change it.

Particular parts of method and object nets, together with the way of their interconnection, are shown in Fig. 2.

2.4 Predicates

PNtalk *predicates* have some features of transitions as well as methods. Predicates are not nets – they do not contain places, and transitions. On the contrary, every predicate can be connected via dialog arcs to some object net places and it also has a guard, which both are typical features of transitions. However, at the same time predicates are equipped with messages in the same way as methods and can be invoked from transition and predicate guards.

Predicates have been introduced into OOPNs to allow atomic testing of objects states. They can be invoked with all parameters well known – then their result is either true or false. Besides, there is also a possibility of using free variables in places of some arguments. In this case, predicates can be either not satisfied or satisfied for different possible bindings of free variables.

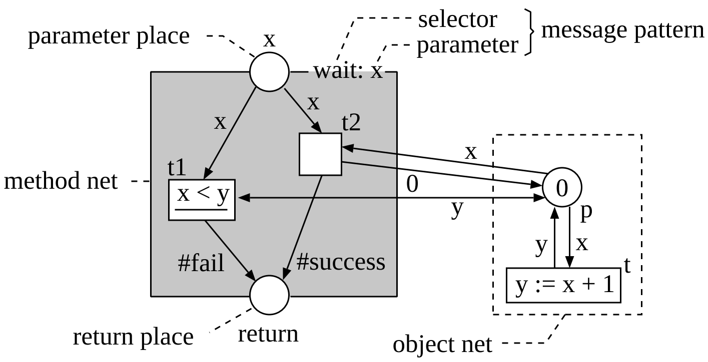

Figure 2: An example of interconnected method and object nets

The predicate in Fig. 3 is satisfied iff there is at least one token in the place `p`. In this case, `x` can be bounded to the object carried by an arbitrary token in the place `p`.

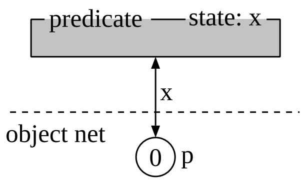

Figure 3: An example of an OOPN predicate

2.5 Constructors

Constructors are special methods which are used for creating and initializing new objects. They look syntactically exactly the same as regular methods, only the key word `constructor` is added.

The fact of physical creation of a new object is not described in constructors – it is realized by an implicit method `new`, which is understood by all classes. Constructors only contain description of an initialization of a newly created object. Messages connected to constructors can be sent either to classes or objects. In the case when a class is the receiver, a new object is created, and then it is initialized. In the latter case, constructors behave like regular methods.

There is a simple example of a constructor in Fig. 4.

2.6 Classes and Inheritance

An OOPN consists of a set of *classes*. One of them is declared as the *primary class*. This class is used for initiating the evolution of the OOPN in such a way that one its instance is automatically created.

A class consists of an object net, a set of method nets and constructors and a set of predicates. An example of two such classes constituting an OOPN is shown in Fig. 5.

With respect to the Fig. 5, it is already possible to guess, that every OOPN class has exactly one ancestor in the inheritance tree. The top of the OOPN inheritance hierarchy can be viewed as a special class which we will call *PN*. The class PN itself is not described by means of Petri nets and it is a descendant of another special class – *Object*. This class is an ancestor of PN and also of so called *primitive classes* whose objects can be attached to OOPN tokens in the same

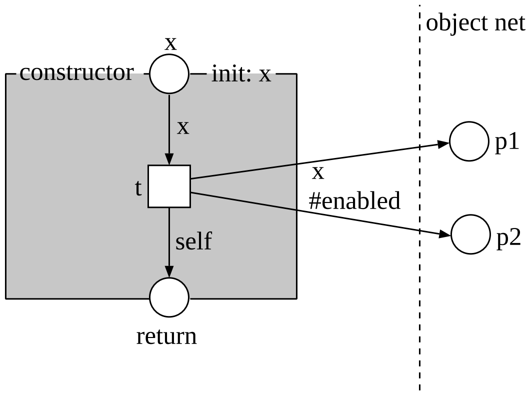

Figure 4: A simple OOPN constructor

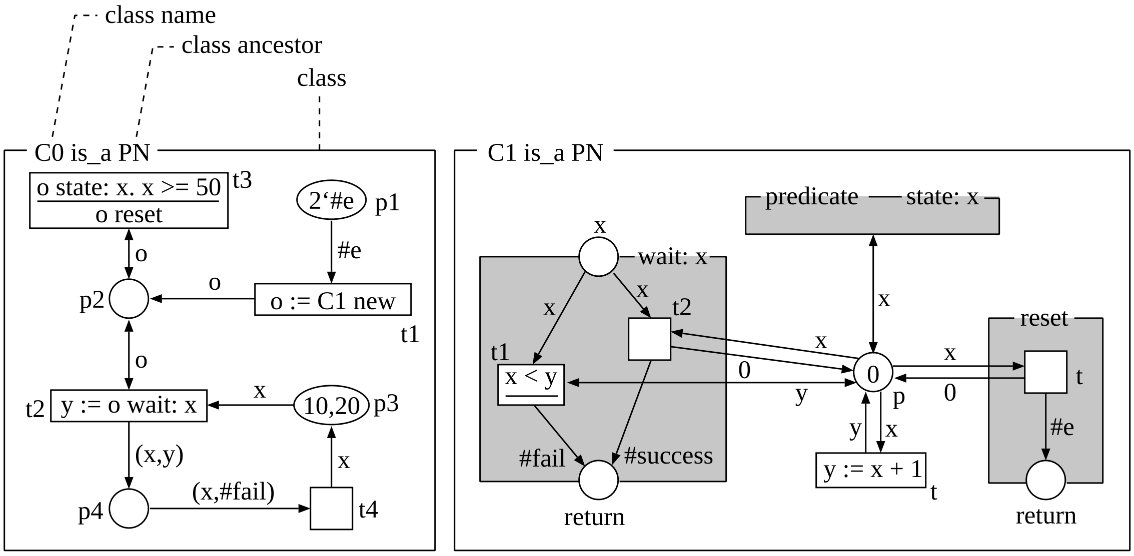

Figure 5: An OOPN consisting of two simple classes (primary class is not specified)

way as regular OOPN objects described by Petri nets. Objects of primitive classes correspond to the literals of the inscription language (see 2.1). The described inheritance hierarchy is depicted in Fig. 6.

Every class inherits its ancestor's own activity, i.e. the structure of the object net, as well as all methods, constructors, and predicates. At the same time everything what is inherited may be redefined. Methods, constructors and predicates can be changed by introducing new methods, constructors, or predicates with the same message selectors as the original ones are. Object nets can be redefined partially by redefining particular transitions and places which are identified again by their names[1]. It is essential to note, that if a transition is being redefined, all original arcs are left out. However, this is not true for redefined places which keep all the inherited arcs together with the new ones.

Apart from redefining, new methods, constructors and predicates can be added as well as

[1]The formal definition of OOPN [12] allows this refinement of net structure also in the case of method nets. However, this mechanism has not been introduced into the PNtalk language yet.

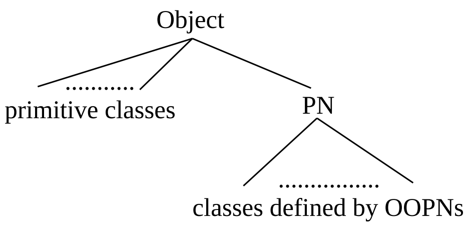

Figure 6: The OOPN inheritance hierarchy

places and transitions in object nets. These new class elements can be of course connected to the inherited or redefined elements.

An illustration of inheritance and redefining is shown in Fig. 7.

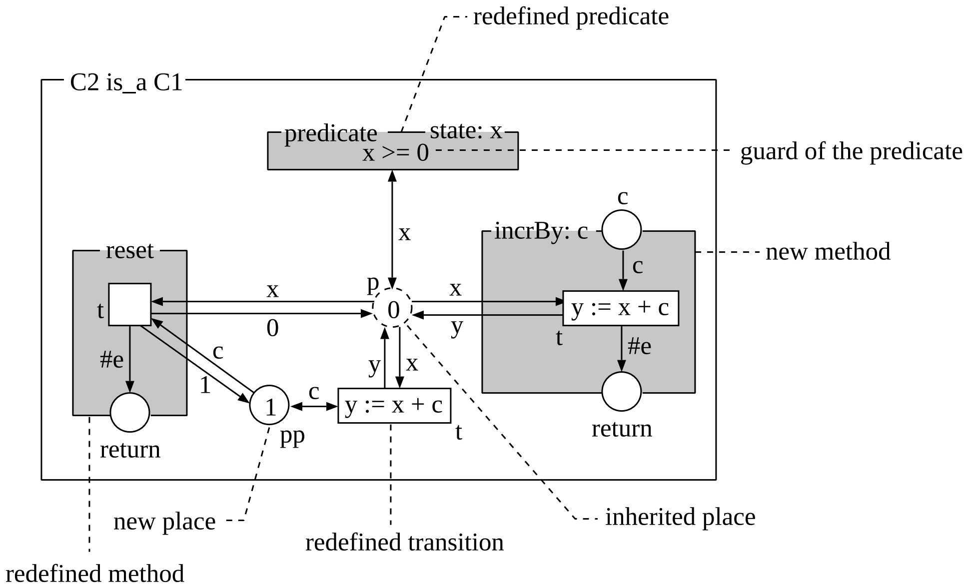

Figure 7: An illustration of inheritance and redefining on a descendant of the class `C1` from Fig. 5. Inherited methods and predicates are not shown.

3 Dynamic Behaviour of OOPNs

The OOPN dynamics is based upon a modified concept of invocation [1]. We have connected the original concept to message sending and extended it with a possibility of polymorphism in the sense that the particular net which shall be invoked is not determined before the moment of the invocation, i.e. it is unknown at the compile time, and at the invocation time it is established according to the class of the message receiver. Apart from the invocation mechanism, nets instances communication by means of shared places is also used here. More precisely, instances of method nets share places with the corresponding object nets. In this way, methods instances can modify the state of appropriate objects.

Transition actions correspond to message sendings. In the following explanation we will consider only actions which consist of one message sending whose result is stored to exactly one variable. However, it is not difficult to extend the transition semantics so that it will deal with sequences of message sendings. Intuitively, such transitions are equivalent to chains of places and transitions with actions containing single message sendings only. Without any respect to the concrete syntax, we will describe transitions actions in the form $y := x_0.msg(x_1, \ldots, x_n)$ where y is a variable, x_i is a term, and msg is a message selector. The object identified by x_0 is the receiver of the message and objects x_i are message parameters. The way how a transition will be performed depends on the class of the receiver, i.e. on the class of the object x_0:

- Some objects are instances of *primitive classes* (see 2.6). Their methods are functions without any effect to the state of the appropriate objects[2]. If x_0 is an object of such a class, the transition together with its action is performed atomically in the same way as in a classical Petri net.
- Another objects are *classes*. They are special objects created at the compile time. These objects do not change during the OOPN evolution. If x_0 is a class, and the message selector is *new*, the transition is performed atomically and a new object of x_0 is created. The identification of this new object is stored to y. The new object contains a new copy of the object net of x_0 which is initialized with the initial marking. If there are non-empty initial actions connected to some places, they can be evaluated as if they were actions of initially fireable transitions storing initial marking to particular places. If a constructor is used instead of *new*, a new object is created by means of *new* and this is initialized by sending it the constructor message which is used in the transition action. It is worth a remark, that the last two concepts need not to be included in the formal definition and can be introduced as the so called "syntactical sugare".
- The remaining objects are instances of classes defined by OOPNs. These objects are considered to be *non-primitive*. If x_0 is a non-primitive object, the transition is performed non-atomically in this way:

  1. The input part of the transition is performed, i.e. tokens corresponding to the binding of the variables of the input arcs are removed. A method corresponding to the selector msg is found according to the class of x_0, a new instance of the appropriate method net is created, and initialized in the same way as an object net instance and finally tokens representing message parameters are added into the method parameter places.
  2. The transition is waiting for the invoked method to end. Meanwhile any transition can be performed, including transitions which have been started but not yet finished, i.e. transitions can be being performed concurrently more than only one times. Particular non-atomic transition executions are called *invocations*. Every invocation can be identified by a tuple (id, b) where id is an identification of the invoked method net instance and b is the binding of the transition variables.
  3. The transition invocation can be finished as soon as some tokens are placed into the return place of the invoked method net instance. Then one object referred to by a token in the return place is chosen, it is assigned to the variable y, the method net instance is deleted, and the output part of the transition is performed, i.e. particular tokens are added into the transition output places.

[2]In the practical implementation of PNtalk there is a possibility of using primitive classes with methods which change the state of the appropriate primitive objects, too. However, their usage makes it impossible even to think about a formal analysis, and it can also slow down the simulation (see 5.3). The semantics of the transition firing in this case is similar to the one connected to non-primitive message receivers.

Up to now we have discussed only transition firing, but of course, transitions can be fired only if they are fireable. A transition is *fireable* for some binding of variables which are present in the arc expressions of its input arcs and in its guard expression, if there are enough tokens in the input places with respect to the values of input arc expressions and if the guard expression for the given binding evaluates to true. It is obvious, that at a given moment a transition can be fireable for more than one binding according to the contents of its input places and to contents of the places (of possible different objects) which can be tested by the predicates called from the transition guard expression.

An OOPN evolution begins by an automatic creation of the so called *primary object* which is an instance of the primary class. This object contains an instance of the particular object net which is initialized by the initial marking. All OOPN dynamics is then given by performing transitions in running net instances which can also influence the set of existing objects and net instances. If we neglect the case of non-atomic methods of primitive objects which are not taken into account in the formal OOPN definition, all evolution steps have the form of atomic *events*. These events are connected to transitions in the following way:

- An atomic transition firing represents a single event of one of the two possible kinds:
  - *A-event.* It is an atomic transition execution inside one net instance. (If this net instance is a method net instance, it can share places with the appropriate object net instance.) This mechanism corresponds to the way of transition firing which is common in common coloured Petri nets.
  - *N-event.* It describes a new object creation by the transition[3].
- A non-atomic transition firing composes a causally connected pair of events:
  - *F-event.* This event represents an execution of the transition input part and a creation of a new method net instance.
  - *J-event.* It deletes the above created method net instance and performs the transition output part.

  Between these two events the transition stores information about the unfinished invocation as a part of its marking, i.e. transitions are marked by unfinished invocations.

An implicit part of every event occurrence is the so called *garbage collecting*, i.e. shredding all net instances as well as objects which are not transitively accessible from the primary object.

A *state* of the system corresponding to a running OOPN can be defined as a set of objects composed of running net instances, whose states are given by their markings. A marking m of a net instance (id, m), where id is an unambiguous identification of the net instance, specifies for every place a multiset of tokens referring to objects, and for every transition a set of invocations.

4 An Example – Patients and Doctors

There is a little bit more complex example given in this section. The ideas of inheritance, active objects and inter-object communication are all included in this example. It is a simulation model of a society of patients and doctors. Because of lack of space, the model was rather simplified, but it is no problem to extend it so that it will become more realistic.

[3]This is the case of using the implicit method *new*. If a constructor method is used, a combination of N-, F-, and J-events arises.

In our model, patients are represented by objects of the class `Patient` (Fig. 8). New patients can be created by the constructor "`with: doctor`", which also specifies for every patient his doctor. The life cycle of patients is described by their object net. We can see that a patient is initially healthy. Then he falls ill and asks his doctor to help him. This is done by sending the doctor a message with the selector `help:`. After having asked the doctor to help him, our patient has to wait until the doctor examines him and tells him what to do to become again healthy. In our case, doctors tell the patients only how long they must wait[4] before they become healthy. Subsequently, the patients are again healthy for some time, and then the lifecycle is repeated.

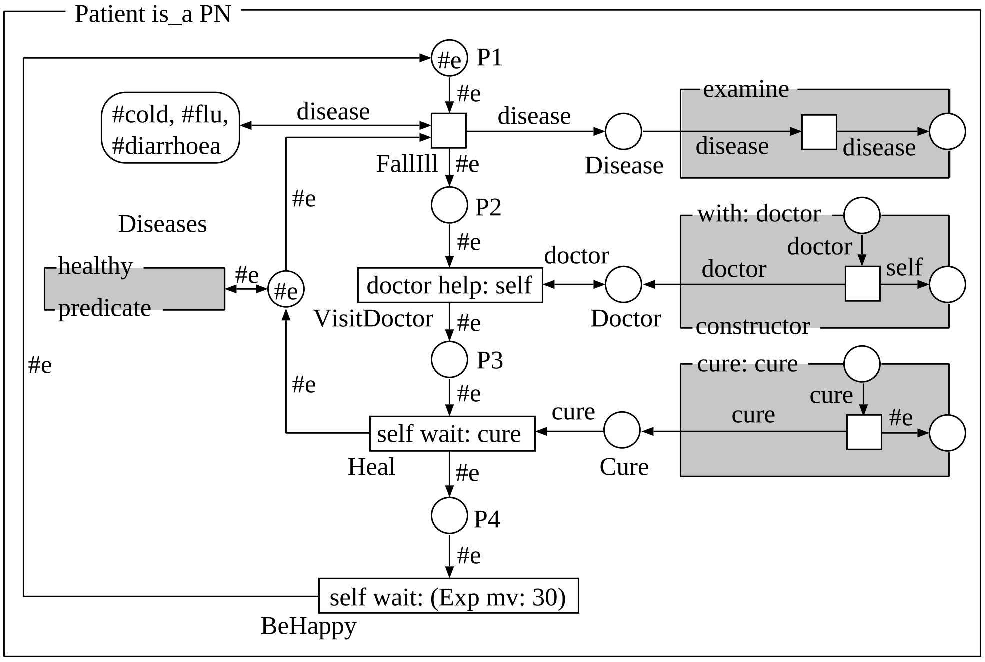

Figure 8: An OOPN class describing patients

A doctor is a special case of a patient – he can fall ill and recover from the illness, but, what is more, he can cure himself and also other patients. Therefore the class `Doctor` (Fig. 9) is a descendant of the class `Patient`. Doctors inherit the own behaviour of patients and modify it in the sense that they do not require any help from another doctors, because they can examine and cure themselves. This modification of patients behaviour concerns the redefined transition `VisitDoctor` and the new transition `SelfCure`. Apart from the inherited lifecycle, the class `Doctor` contains another cycle which is connected to the work of doctors. If a doctor is healthy and some patient announced him an illness, the doctor examines the patient, i.e. sends him the message `examine`, then he finds out how long the patient will have to wait to become again healthy, and announces it to the patient. Then this cycle is repeated.

The class `PatientsAndDoctorsWorld` (Fig. 10) represents the top layer of our model. This class is used as the primary class of the OOPN. It states that there will be five patients and those will be treated by two doctors.

[4]There is implemented the mechanism of the *firing time* by means of the method `wait:` of the class `PN` in PNtalk. It works in the following way: Input tokens are removed and output are produced after the time specified by the argument of `wait:` has passed. Currently we plan to introduce also the concept of the *enabling time* into PNtalk. More details about mentioned time concepts can be found e.g. in [7, 8, 10].

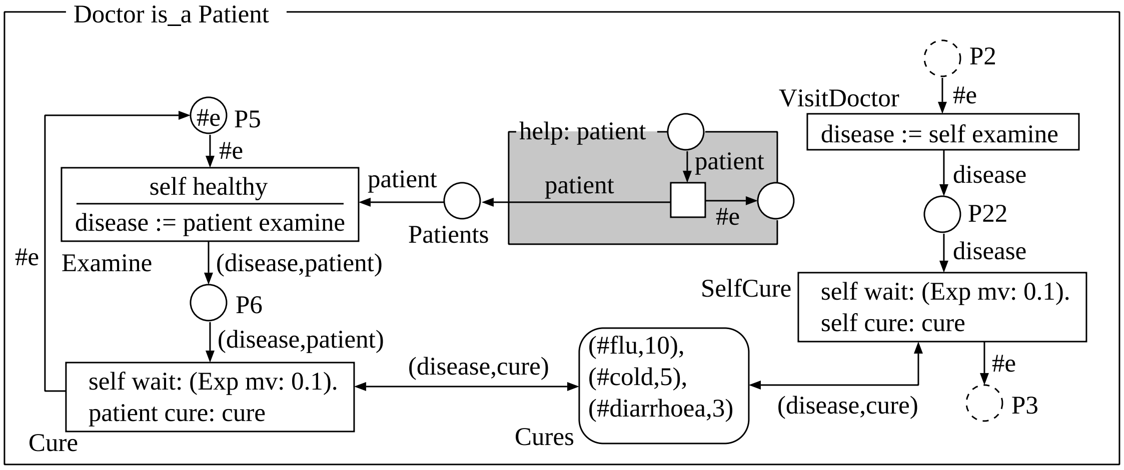

Figure 9: An OOPN class describing doctors as special cases of patients

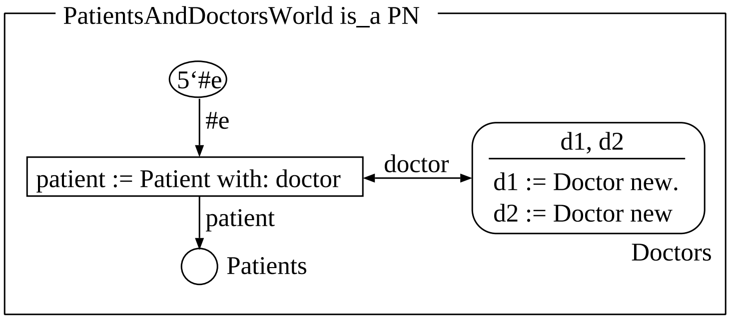

Figure 10: An OOPN class describing a world of doctors and patients

It is obvious that our model is rather unrealistic – patients and doctors enjoy eternal life, patients wait for their doctors without any limits of waiting times, doctors cannot fall ill in the middle of their work, and so on. It is possible to improve the model, however this is beyond the scope of this article.

5 The PNtalk System

We have already seen that *PNtalk* is a concurrent object-oriented language based on high-level Petri nets and inspired by Smalltalk. It is an experimental language intended for formal specifications, simulation modelling and programming. *PNtalk system* is a computer tool designed to support the use of object oriented Petri nets connected to the language PNtalk.

In fact, the PNtalk system is an additional tool incorporated into the ParcPlace VisualWorks Smalltalk environment. This environment allows us to write Smalltalk classes, create graphical interfaces, communicate with databases etc. All these features can be used from the object oriented Petri nets as well. Nevertheless, it is of course possible to work with Petri nets only. In this case, the VisualWorks system is used only for launching the PNtalk system.

5.1 PNtalk Launcher

The so called PNtalk Launcher (Fig. 11) is used as a control panel – it is possible to start all other tools from here. In particular, a user can launch a browser, a debugger or a tool for manipulation with breakpoints. It is also possible to start multiple browsers or debuggers for working with various parts of the modelled system.

The launcher also contains buttons for OOPN class system compiling, for adjusting various simulation options, and, of course, for starting and terminating simulation runs. Before starting a particular simulation run, the user must specify the main (primary) class in the launcher. This will be used when starting the simulation in such a way, that the simulator will automatically create one instance of the main class.

Further, the PNtalk launcher is used for displaying the present status of the PNtalk system, and also for displaying current simulation time during simulation runs. It also contains the so called *transcipt* which is used for displaying various system messages as well as user defined messages which can be sent from transition actions.

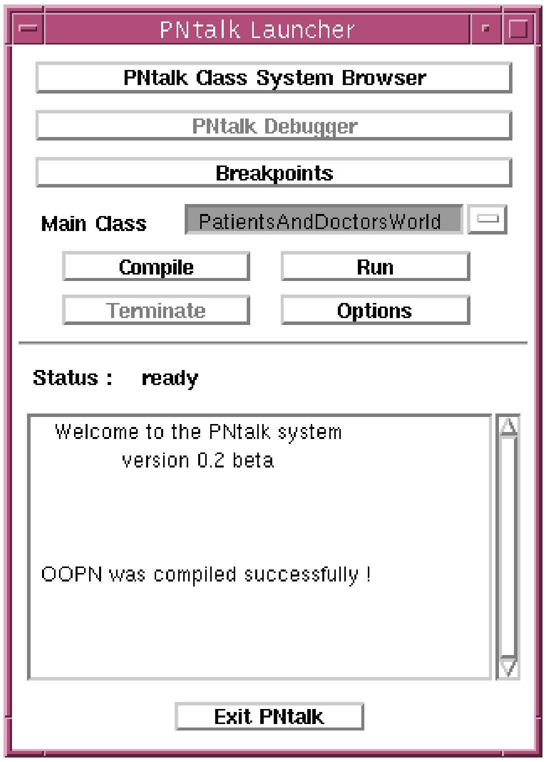

Figure 11: PNtalk Launcher – a control panel of the PNtalk system

5.2 PNtalk Browser

The tool of PNtalk designed for editing OOPNs is called PNtalk Browser. Its window (Fig. 12) is divided into three parts. The first part contains a list of OOPN classes, the second a list of methods and constructors of a particular class chosen in the first part. The third and the biggest part is intended for displaying and editing the graphical representation of object nets and method nets.

The browser is controlled by a system of menus which can be invoked by mouse clicks, and which depend on the position of cursor and the chosen graphical object. The menu invoked on

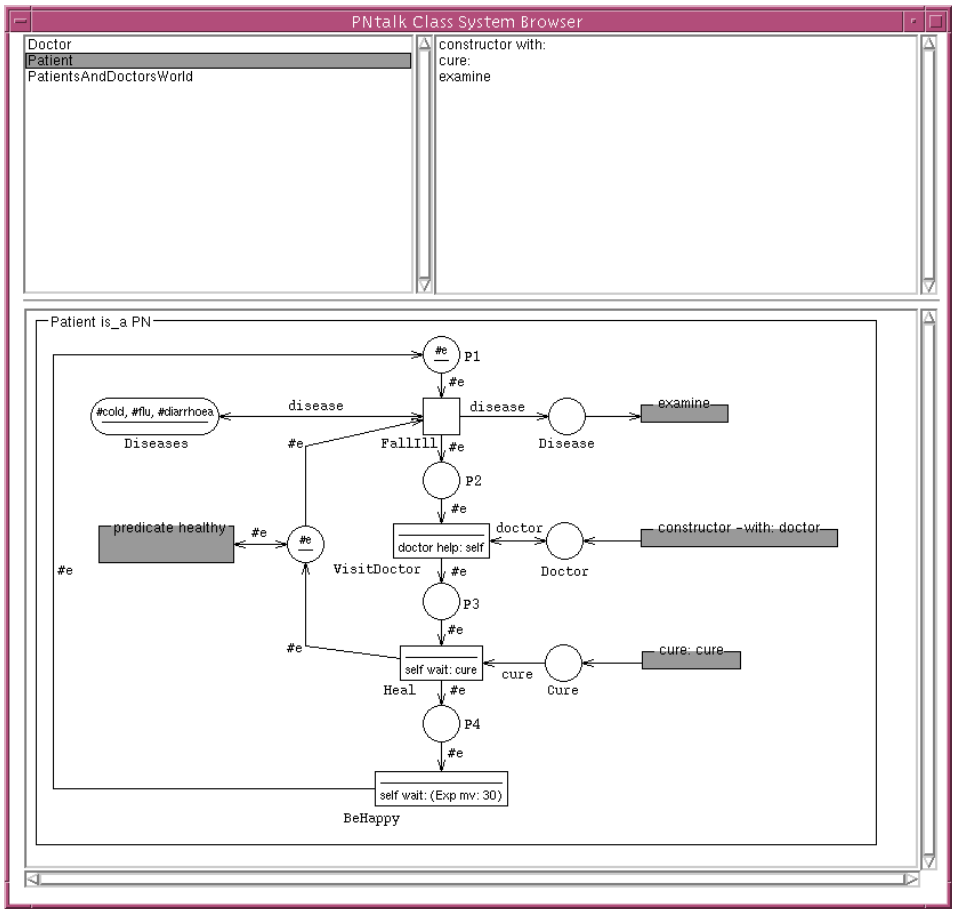

Figure 12: PNtalk Browser – a tool for creating, displaying and modifying OOPNs

the list of classes can be used for adding new classes, removing classes, loading and saving class hierarchies or printing. Similarly, menu invoked on the list of methods can be used for creating and destroying methods, constructors and predicates. When invoking menu in the graphical part of the browser, it is important what a graphical object has been chosen, e.g. place, transition, or others. For example, the place menu allows us to edit the place initial marking, action or name, we can add new arcs going from the chosen place, and so on.

5.3 Various Modes of OOPNs Simulation in PNtalk

Before starting a simulation run a number of parameters must be specified. These parameters are defined in the window which we can see in Fig. 13.

The first group of parameters say which *simulaton mode* will be used, i.e. how the user will be able to control the simulation run. We distinguish the following simulation modes:

1. *Automatic mode.* The user does not have a possibility to affect the simulation run; all transition instances are fired automatically. We can distinguish two submodes here:
   (a) *Normal automatic mode.* Firstly all fireable transitions and transition bindings are found, and then one transition binding is chosen and fired.
   (b) *Fast automatic mode.* As soon as the simulator, when checking bindings fireability, finds one fireable binding, it immediately fires this binding.

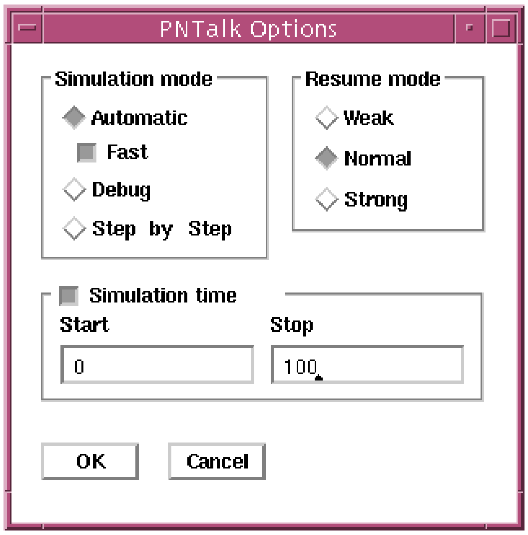

Figure 13: Various simulation modes of OOPNs in PNtalk

2. *Step-by-step mode.* In this case, the simulator collects all fireable bindings, those are shown to the modeller, who chooses which one will be fired.

3. *Debug mode.* The debug mode is exactly the same as the normal automatic mode, except for the fact that breakpoints defined by the user in the browser are taken into account.

The next group of options specifies the so called *resume mode.* To be able to describe it, we must review conditions under which transition fireability can be changed:

1. Some tokens are added to or removed from an input place of the transition.

2. The state of an object bound to a token in an input place is changed. The change can concern either an OOPN object or a Smalltalk object.

3. The state of the object in which the transition instance is situated is changed. This state can be tested by means of `self` or `super`.

Generally, it is necessary to test transition fireability of affected transitions after any change from the above list. The same result can be achieved by testing the fireability of all transition instances before every firing. This method is implemented in the *strong resume mode*, but it is of course rather time consuming.

When we use PNtalk for modelling based on P/T Petri nets or coloured Petri nets without using Smalltalk objects whose state can be changed, only the first item of the above list must be taken into account. This is the case of the so called *weak resume mode.*

Finally, the *normal resume mode* includes the entire weak mode, and, what is more, it can detect changes of transition fireability caused by state changes of OOPN objects in input places. This is true even if the changed OOPN object is stored in a PNtalk list. This mode can also detect a state change of the object in which the particular affected transition instance is situated. So, the only difference between this mode and the strong one is that it is unable to detect fireability changes caused by changes of Smalltalk objects state.

The last think which can be specified in the options window is, whether the *simulation time* will be used and if so, what will be its ranges.

5.4 PNtalk Debugger

The window of the PNtalk debugging tool (Fig. 14) is again divided into three parts. There is a list of created objects and methods instances working over them in the left-most part. Spacing is used here to show which method called which. In this way, a process tree is created. The middle part displays the graphical representation of the chosen net instance. A place or a transition highlighted in this part specifies then the contents of the last, right-most part. In the case of a place, its current marking is shown here. In the case of a transition, its possible bindings are displayed, together with the bindings for which the transition action has already been started but not yet finished. The number of fireable and processed bindings is summarized in the top right corner in the form `fireable bindings/processed bindings`.

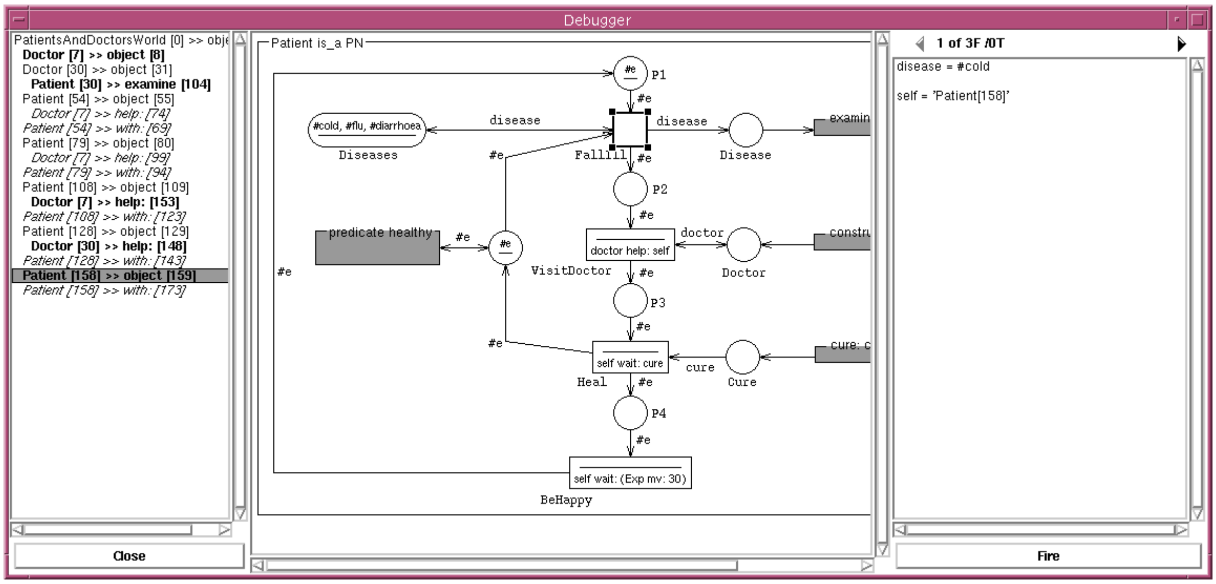

Figure 14: PNtalk Debugger – a tool allowing visualized token game

If an OOPN object is a part of a binding, a menu can be launched on the appropriate item in the right part of the debugger, which can be used for straight move to the description of the state of this object. It is also possible to open more then one debugger and to watch the evolution of several nets in a concurrent way.

Transition instances which can be fired are drawn by thick lines in the graphical representation of nets. Similarly, net instances with fireable transitions are written in bold in the list of methods and objects.

There is also emphasized by means of italics in the list of objects and methods which objects or methods are already dead, i.e. were destroyed or finished. This items can be hidden from the menu which can be launched in the left part of the debugger. What is more, a garbage collector can be invoked by an explicit command here, and all obsolete items can be shred.

At the end of this subsection, it is necessary to mention the so called *postmortem debugger*. This debugger can be invoked after finished or interrupted simulation for examining the terminal state of the simulated OOPN. This is impossible only in the case of using the fast automatic mode.

6 Conclusions

This paper presents an alternative approach to combining Petri nets and the object oriented modelling paradigm. The resulting formalism describes systems by sets of classes of active objects. Both own activities of objects as well as their responses to messages are described by Petri nets. All advantages of the object oriented paradigm, e.g. inheritance, polymorphism, and encapsulation, have been adapted for the Petri net environment. At the same time the proposed paradigm can still be formally defined and it is possible to think about formal analysis of models based on it.

The described OOPN formalism serves as a base of the language and system PNtalk. The prototype implementation of the system PNtalk in the Smalltalk environment has also been mentioned here. Currently this system cannot be used for working with large models, but there is an intention to implement a new and more powerful version of this tool soon.

In the future we plan to introduce also classical structuring techniques into the pure OOPN formalism. Thus the modeller will be able to choose which aspects of the modelled system will be described by independent communicating objects, and which will correspond to nets (or pages [2, 6]) statically interconnected by socket and port nodes. We also intend to properly introduce time into the OOPNs. For this purpose, the flexibility offered by the object orientation will be exploited in such a way, that the user will be able to work either with timed Petri nets [5] or stochastic Petri nets [7]. Finally, we want to verify how the proposed OOPN formalism could be practically applied in various areas like modelling flexible manufacturing systems, real-time systems, or workflow and information systems.

Acknowledgment

This work was supported by the grant of the Czech Grant Agency GAČR No. 102/96/0986 and by the Universities Development Fund of the Czech Ministry for Education under the contract F1: 618/1997.

References

[1] Huber, P., Jensen, K., Shapiro, R.M.: *Hierarchies in Coloured Petri Nets.* In: Advances in Petri Nets 1990, Lecture Notes in Computer Science 483, Springer-Verlag, 1991.

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

[3] Lakos, C.A., Keen, C.D.: *LOOPN++: A New Language for Object-Oriented Petri Nets.* Technical Report R94-4, University of Tasmania, 1994.

[4] Sibertin-Blanc, C.: *Cooperative Nets.* In: Application and Theory of Petri Nets 1994, Springer-Verlag, 1994.

[5] Ghezzi, C., Morasca, S., Pezzè, M.: *Timing Analysis of Time Basic Nets.* The Journal of Systems and Software, 1994.

[6] Jensen, K.: *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use, Volume 2.* Springer-Verlag, 1995.

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

[8] Schöf, S., Sonnenschein, M., Wieting, R.: *High-level Modeling with THORNs.* In: Proceedings of the 14th International Congress on Cybernetics, Namur, Belgium, 1995.

[9] Janoušek, V.: *PNtalk: Object Orientation in Petri Nets.* In: Proceedings of the European Simulation Multiconference ESM'95, 1995, Prague Technical University, Prague, Czech Republic, 1995.

[10] Bowden, F.D.J.: *Modelling Time in Petri Nets.* In: Proceedings of the Second Australia-Japan Workshop on Stochastic Models, Gold Coast, 1996.

[11] Češka, M., Janoušek, V.: *Object Orientation in Petri Nets.* In: Object Oriented Modelling and Simulation. Proceedings of the 22nd Conference of the ASU, University Blaise Pascal, Clermont-Ferrand, France, 1996.

[12] Janoušek, V.: *Objektově orientované Petriho sítě – formální základ jazyka PNtalk.* Technická zpráva, ÚIVT FEI VUT v Brně, 1997. (in czech)

[13] Češka, M., Janoušek, V.: *A Formal Model for Object Oriented Petri Nets Modeling.* In: Advances in Systems Science and Applications, ASSA Special Issue, International Institute for General Systems Studies, Southwest Texas State University, San Marcos, Texas, USA, 1997.

[14] Esser, R.: *An Object Oriented Language for Emdedded System Design.* In: Proceedings of 8th International Workshop on Software Technology and Engineering Practice, London, 1997.