On the Dynamic Features of PNtalk

Radek Kočí and Vladimír Janoušek

Brno University of Technology,
Bozetechova 2, 61266 Brno, Czech Republic
{koci,janousek}@fit.vutbr.cz
http://www.fit.vutbr.cz/

Abstract. PNtalk is a tool based on Object Oriented Petri nets. It is intended for systems modeling, simulation and prototyping. In some situations, it is also possible to use it as a programming language and a framework for final implementation of the systems. The paper presents a meta-level architecture of the PNtalk kernel and and demonstrates its reflective features. These features are crucial for the systems development process as well as for the systems maintenance. The usage of the PNtalk metaobjects are demonstrated by examples.

Key words: Object-oriented Petri nets, meta-level architecture, modeling, simulation, rapid prototyping

1 Introduction

PNtalk is a tool for modeling, simulation, and prototyping complex systems. It is based on a formalism called Object Oriented Petri Nets (OOPN). OOPN [8] combine advantages of object orientation and Petri nets. The OOPN formalism is based on high level Petri nets allowing to describe the work-flow and parallelism in the systems. Object orientation of the OOPN formalism allows for better structuring of the models, which is conform with current software development methodologies. OOPN define objects in a very similar way like other object oriented languages but with one important difference – the methods are not described as the 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.

The current version of the PNtalk tool allows for a higher level of dynamism and a higher level of control over the OOPN interpretation. 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 obtain 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.

To achieve features described above, the open implementation issue including the reflective and meta-level architectures was taken into account. The feasibility of the open implementation approach depends on the degree of its 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 the system as open as possible. The paper does not bring complex case study featurign reflection. It rather concentrates on the explanation of main architectural features using simple examples.

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 older (Lisp, Universal Turing machine). As the examples of recent activities the following projects can be cited – Kiczales [12] and Maes [16] introducing aspect-oriented programming, Actalk [3] introducing concurrency via reflection in Smalltalk, Apertos [22] and TUNES [1] 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 [20] introducing reflectivity to DEVS [23] in order to allow for structural changes of models. These approaches are important especially for modeling and simulation of intelligent agents.

Our attempts to incorporate reflection to the Petri nets also are not alone. Lakos [15] 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.

The paper is organized as follows. First, we briefly describe the OOPN formalism and its important elements. The third chapter deals with the basic principle of the PNtalk architecture. The next three chapters describe the main architectural elements and features of the OOPN classes, OOPN objects, including simulation, and the inter-object communication at different levels.

2 Object Oriented Petri Nets

A lot of attempts to combine Petri nets and object-orientation has been done since 1980s. The probably best known issues have been introduced by Lakos [14], Sibertin-Blanc [19], Moldt [17], and Valk [21]. One of the approaches is a formalism called Object Oriented Petri nets (OOPN) and PNtalk language and system that was developed in 1994 and published in [7, 4]. It combines pure object-orientation inspired by Smalltalk [5] with high-level Petri nets.

Following the Smalltalk-like style, all objects are instances of classes, every computation is realized by message sending, and variables can contain references to objects. A class defines structure and behavior of its instances. 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 can represent concurrency inside a method or object net).

An Object Oriented Petri Net (OOPN) consists of Petri nets organized in classes. Each *class* consists of an *object net* and a set of dynamically instantiable *method nets*. Places of the object net are accessible for the transitions of the method nets. Object nets as well as the method nets can be inherited. Inherited transitions and places of the object nets (identified by their names) can be redefined and new places and/or transitions can be added in subclasses. Inherited methods can be redefined and new methods can be added in subclasses. Classes can also define special methods called *synchronous ports*, which allow for synchronous interactions of objects. Message sendings and object creations are specified as actions attached to transitions. The transition execution is polymorphic — the method which has to be invoked is chosen according to the class of the message receiver that is unknown at the compile time. A token in a place represents either a *primitive object* (e.g., a number or a string) or an instance of an OOPN class. The instance consists of an instance of the appropriate object net and possibly several concurrently running instances of the invoked method nets.

The transition guards and actions can send *messages* to objects. The way how transitions are executed depends on the *transition actions*. A message that is sent to a primitive object is evaluated atomically (thus the transition is 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. Each method net has parameter places and a return place. These places are used for passing data (object references) between the calling transition and the method net.

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. Synchronous ports 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. A special variant of the synchronous port concept is *negative predicate*. Its semantics is inverted—the calling transition is firable if the negative predicate is not firable.

An example illustrating the important elements of the OOPN formalism is shown in Figure 1. Two classes *C0* and *C1* are depicted there. The object net of the class *C0* consists of places *p1* and *p2* and one transition *t1*. The object net

of the class *C1* is empty. The class *C0* has a method *init:*, a synchronous port *get:*, and a negative predicate *empty*. The class *C1* has the method *doFor:*.

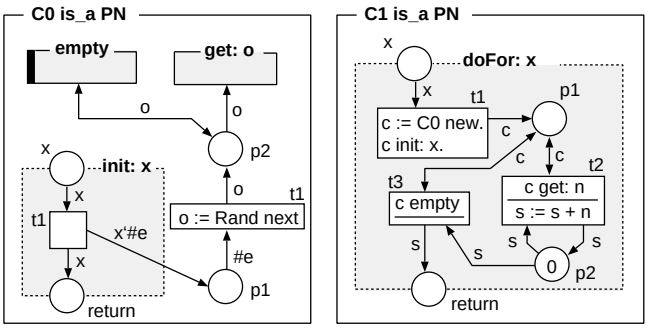


Fig. 1. An OOPN example.

Let us have an expression *C1 new doFor: 3*. Its execution leads to the creation of an instance (object o) of $C1$ and an instance of *doFor:* belonging to the object o. When $t1$ belonging to the instance of *doFor:* inside the object o leads to the instantiation of $C0$ (let the instance be named o') and *init:* inside the object o'. Then, $t1$ inside o' can be executed for three times. Consequently, $t2$ inside the instance of *doFor:* inside o can be executed. Its guard invokes synchronous port *get* : of o'. The variable n is boud to the value of the token from the place $p2$ belonging to o'. When the place $p2$ of o' is emptied, $t3$ belonging to the instance of *doFor* : inside the object o can be fired because the negative predicate *empty* of o' is satisfied.

Each OOPN model has its text representation (source code). It is very important for the meta level manipulation with models. We will demonstrate it on the previous example. The Figure 2 shows the text representation of the class C0, its object net (the keyword `object`), its synchronous port (the keyword `sync`), its negative port (the keyword `negative`), and its method net (the keyword `method`).

Every transition, synchronous port, and negative predicate define its preconditions, conditions, and postconditions. For instance, the synchronous port `get: o` is conditioned by the precondition `precond p2(o)`. It means, that the port is firable if the place contains at least one object and if the port will be fired it will remove this object from the place `p2`. In addition, transitions and ports can have guards (the keyword `guard`) and actions (the keyword `action`) as we can see in the Figure 3 of the class `C1` source code.

```
class C0 is_a PN
  object
    place p1()
    place p2()
    trans t1
      precond p1(#e)
      action{o := Rand next}
      postcond p2(o)
  sync get: o
    precond p2(o)
  negative empty
    cond p2(o)
  method init: x
    place x()
    place return()
    trans t
      precond x(x)
      postcond p1(x‘#e), return(x)
```

Fig. 2. The source code of the class `C0`.

```
class C1 is_a PN
  method doFor: x
    place x()
    place p1()
    place p2(0)
    place return()
    trans t1
      precond x(x)
      action {
        c := C0 new.
        c init: x.}
      postcond p1(c)
    trans t2
      precond p2(s)
      cond p1(c)
      guard {c get: n}
      action {s := s + n.}
      postcond p2(s)
    trans t3
      precond p2(s)
      cond p1(c)
      guard {c empty}
      postcond return(s)
```

Fig. 3. The source code of the class `C1`.

3 PNtalk System Architecture

PNtalk (*P*etri *N*et *talk*) is the tool based on the formalism of OOPN. Its purpose is to make a framework for experiments with simulations as well as formal approaches to the system design [10, 11]. Both OOPN and PNtalk are closely associated with the Smalltalk environment. Smalltalk is the inscription language of the OOPN formalism (actions and guards are described using Smalltalk) and the PNtalk system is implemented in Smalltalk. The PNtalk system is incorporated into the other experimental tool named *SmallDEVS* [6] which is based on the DEVS formalism [23]. PNtalk uses hierarchical repositories of SmallDEVS to store OOPN classes and allows for joining models described by OOPN and DEVS formalisms.

This chapter discusses basic ideas behind the PNtalk system architecture. It is based on the principles of open implementations [9], namely the meta-level architecture. These principles are also shown on the examples. They are written in the Smalltalk environment, but they can be used inside models too.

3.1 Open implementations

The recent systems for complex application support allow the applications not only to use the services offered by the system, but they also offer means to control how these services are provided and processed. The traditional approach (the black-box abstraction) says that some abstraction (object) should expose its functionality but hide its implementation. It 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 the changing requirements, and/or to develop the applications during their life-time etc. The open implementation principle offers a solution of the problems.

The basic idea of an open implementation is to allow a model to inspect inner aspects of the domain objects (introspection) and to work upon these aspects (reflection). The classic case of an open implementation is the meta-level architecture partitioning a model into two layers – the domain (or basic) level and the meta-level [16]. All the objects describing the domain problem represent the domain level. To each object at the domain level there is a special object (or a set of objects) at the meta-level – metaobject. The meta-level should be understood as a denotation of something what stays behind an object and reflects (or describes) its features and properties—de facto describes information about information. A metaobject offers the metaobject protocol for inspecting and changing the selected aspects of its domain object. The meta-level architecture allows not only to work upon structures of the domain 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.

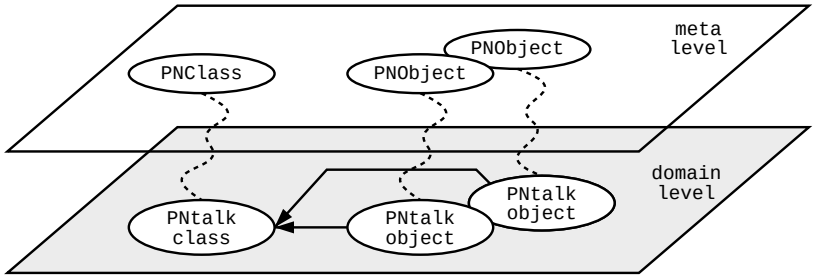


Fig. 4. The PNtalk meta-level architecture – basic overview.

3.2 Meta-level Architecture

The PNtalk architecture introduces a new meta level between the domain objects (i.e., PNtalk classes and PNtalk objects) and Smalltalk. Objects belonging

to the meta-level are implemented by classes of Smalltalk (a basic overview of the PNtalk architecture is presented in Figure 4). While the Smalltalk meta-level architecture is based on classes (i.e., objects are instances of their metaobjects), the PNtalk meta-level architecture is based on objects (i.e., object are not instances of metaobjects, but metaobjects implement the corresponding domain objects).

The PNtalk meta-level comprises metaobjects which control PNtalk classes and PNtalk objects. The metaobjects of the first kind describe *the structure* of PNtalk classes and define the ways of the manipulation with them. The metaobjects of the second kind describe *the computational behavior* of the PNtalk objects (instances of PNtalk classes).

3.3 Metaobjects composition

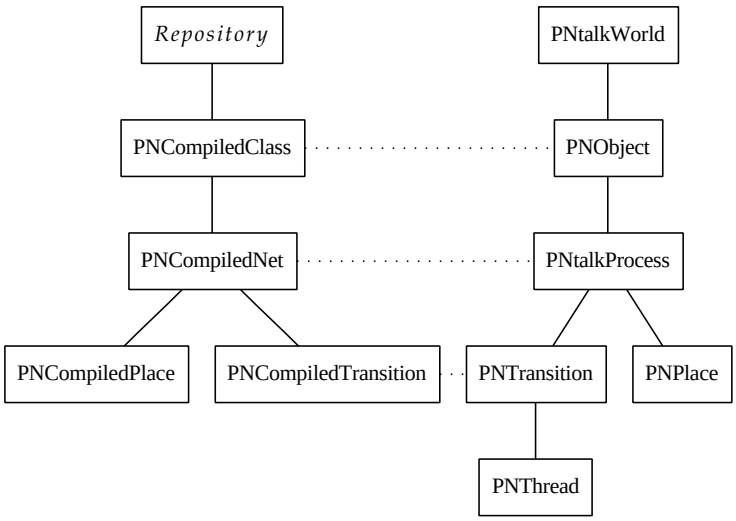


Fig. 5. The basic composition of the metaobjects.

Before we describe the metaobjects and their protocol, we should pay attention to the basic composition of them (see the Figure 5). We will not discuss all parts of the composition, but we will just deal with the most important ones. As we have already said each OOPN class or object is represented by its metaobjects. These metaobjects are instances of the suited classes. These classes are part of the framework implemented, in this case, in Smalltalk. For instance, the OOPN class is represented by the instance of the framework class

`PNCompiledClass` (since the instance is a result of some compilation process we call it with a prefix `Compiled`).[1]

Each OOPN class (the metaobject *PNCompiledClass*) consists of nets (the metaobject *PNCompiledNet*), each net consists of places and transitions, etc. Each OOPN class is placed in a repository which serves as a name space for the domain classes. The metaclass for the repository is specified by italic font in the Figure 5—the repository is a part of the SmallDEVS system to which the PNtalk system is incorporated. This part is not very important for the way how the PNtalk system is explaned here; for more details about this please see [6].

Each OOPN object (the metaobject *PNObject*) is an instance of its OOPN class. In the architecture, the metaobject *PNObject* knows about its class reprtesented by the metaobject *PNCompiledClass*. So that if we send a message to the OOPN object at the domain level, the object looks for the method in the dictionary described by the instance of the class `PNCompiledClass` at the meta level.

Each OOPN object consists of processes (the metaobject *PNProcess* represents an invocaion of a method or an object net), each process consists of transitions (the metaobject *PNTransition*), places (the metaobject *PNPlace*), and thread (the metaobject *PNtalkThread*). Threads represent fired transitions, the transition can be fired for more times simultaneously. The simulation is represented by the metaobject *PNtalkWorld*. Each OOPN object has to be placed into some world in order to be runnable (i.e. can be simulated).

In addition to this, there is the special metaobject accessible via the name *PNtalk* (it is an instance of the class *PNtalkSystem*) supporting special services and requirements (garbage collecting, getting other auxiliary metaobjects etc.). This metaobject is not shown in the presented hierarchy because it is not important for this paper.

4 Representation of the Domain Classes

This section discusses the part of the PNtalk architecture describing the OOPN classes. This part consists of objects (and their classes) representing appropriate elements of OOPN classes (i.e., the *PNCompiledClass* for OOPN class, the *PNCompiledNet* for method net, etc.) and other auxiliary metaclasses. The inheritance hierarchy of classes ot these metaobjects is shown in the Figure 6. The auxiliary class *PNClassComponent* is a root of this inheritance hierarchy and supplies basic services for all other classes of the PNtalk metaobjects. We will deal only with the *PNCompiledClass* in this paper because of two reasons: first, the other classes from this part is not used in our examples, and, second, due to the limited space of this paper.

[1] For the text simplicity, when we will talk about, e.g., *the metaobject `PNCompiledClass`*, we will understand it as an instance of the framework class `PNCompiledClass`.

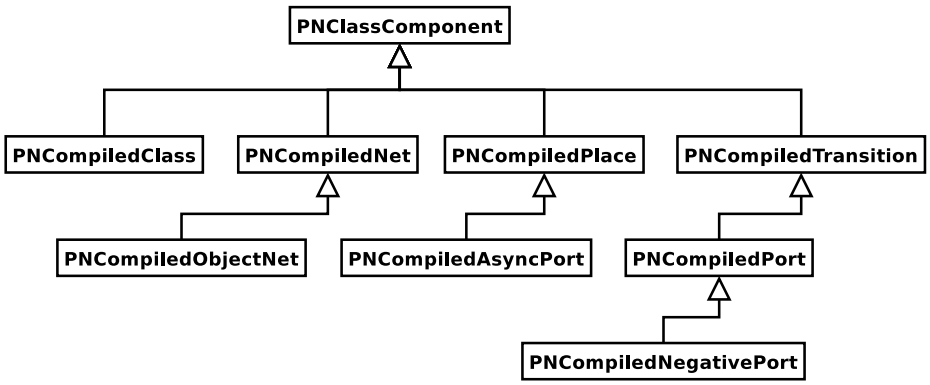


Fig. 6. The inheritance hierarchy of the classes of the OOPN class metaobjects.

PNCompiledClass Instances of *PNCompiledClass* represent the domain classes of the OOPN formalism. The list of selected meta-operations from the metaobject protocol follows:

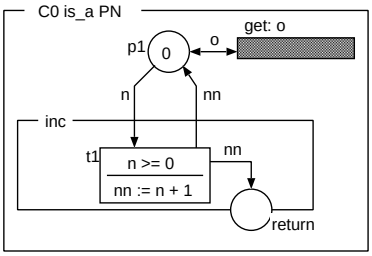
`compile:` compiles a source code of the method (or object) net or synchronous (or negative) port and adds the compiled one (i.e., the instance of the metaclass *PNCompiledNet*, or *PNCompiledPort*, or *PNCompiledNegativePort*) into this PNtalk class
`new` creates the instance of this PNtalk class
`newIn:` creates the instance of this PNtalk class and placed it into the specified simulation space

4.1 Example: Creating the New OOPN Class

Let us have simple example shown in the Figure 7. It demonstrates creating of the new metaobject representing the new PNtalk class `C0`. We can see that OOPN have their text representation (source code) which can be used to the class description and construction. The figure shows the resulting class in graphic notion. This example creates one object net and one method net. The object net consists of the place `p1` and the synchronous port `get:`. The method net consists of the transition `t1` which increments a content of the place `p1` and returns the result (the place `return`).

5 Representation of the Domain Objects and Simulation

This section discusses the part of the PNtalk architecture describing OOPN objects and their simulation. This part consists of the classes of the metaobjects representing appropriate elements of OOPN objects (i.e., the *PNObject* for OOPN object, the *PNProcess* for running method net, etc.) and other auxiliary metaobjects. The inheritance hierarchy of these classes is shown in the Figure




```
cls := PNCompiledClass new.
cls name: 'C0'.
```

```
cls compile: '
object
  place p1(0)
  sync get: o
    cond p1(o)
method inc
  place return()
  trans t1
    precond p1(n)
    guard { n >=0 }
    action { nn := n + 1 }
    postcond p1(nn), return(nn)
'.
```

Fig. 7. The example of creating the new OOPN class.

8. We will deal only with the selected metametaobjects in this paper because of reasons mentioned in the section 4.

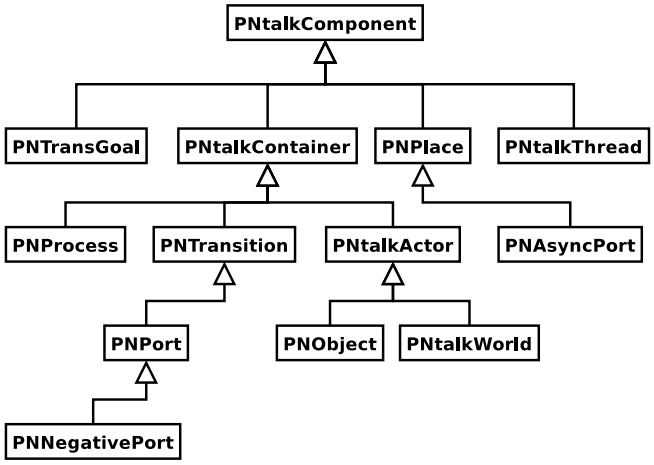


Fig. 8. The inheritance hierarchy of the classes of the metaobjects of the OOPN objects and simulations.

PNtalkComponent The class *PNtalkComponent* is a root of the inheritance hierarchy. It offers basic metaprotocol which is common for all metaobjects. The protocol mainly consists of operations allowing to set or to get the unique object identification (`id`, `id:`), name (`name`, `name:`), and parent object (`parent`,

`parent:`) from the metaobject composition point of view (e.g., a place is a component of some process).

PNPlace Instances of the class *PNPlace* represent places in object or method nets. The list of the selected meta-operations from the metaobject protocol follows:

`add:mult:` adds multiple copies of specified object into the place
`take:mult:` gets and removes multiple copies of specified object from the place
`contains:` tests if the place contains specified object

PNtalkContainer The important metaclass is *PNtalkContainer*. It offers basic metaprotocol for such metaobjects which can store other metaobjects. The metaprotocol allows for adding, removing, and searching of subcomponents. The list of the selected meta-operations from the metaobject protocol follows:

`addComponent:` adds a specified component
`removeComponentNamed:` remove a component identified by the specified name
`componentID:` gets a component identified by the specified id
`componentNamed:` gets a component identified by the specified name
`componentNames` gets a collection of components names

PNProcess Instances of the metaclass *PNProcess* represent evoked method or object nets. The list of selected meta-operations from the metaobject protocol follows:

`placeNamed:` gets a metaobject of *PNPlace* specified by its name
`transitionNamed:` gets a metaobject of *PNTransition* specified by its name

PNtalkWorld The instance of this class represents one simulation space which is called *world*. To be simulated (executed), each OOPN object has to be placed into some world. The simulation algorithms are implemented by this metaobjects in coordination with the metaobject of *PNObject*. The list of selected meta-operations from the metaobject protocol follows:

`start` starts a simulation. This operation is asynchronous and enables inner simulation mechanism. This mechanism calls the *step* operation in the cycle until there is at least one event or if the new one occurs.
`stop` stops a simulation. This operation is asynchronous and disables inner simulation mechanism.
`step` does one simulation step. In each step, it sends the message *step* to the each object having at least one event to perform.
`test` tests transitions and event for firability (needed after every changes from outside).

PNObject Instances of the class *PNObject* represent domain objects, i.e., objects of the OOPN formalism. It offers means for its simulation too. The list of selected meta-operations from the metaobject protocol follows:

`yourClass` gets a metaobject representation (the metaobject *PNCompiledClass*) of the OOPN class.

`compile:` compiles a source code and adds newly created elements (changed object net, method net, ports, etc.) into the object. These changes do not take effect in the OOPN class but only in this instance (an OOPN object).

`performDomainMessage:` performs a domain message. The message is in a special form (special metaobject). The operation is asynchronous—it looks for appropriate method net, creates its instance, i.e., the process as an instance of the metaobject *PNProcess*, and returns. The process execution is then under the control of *PNObject* and is independent of the other processes. At the domain level, the message sending is synchronous, i.e., the calling thread or object waits until this process finished.

`testPort:` tests a port. The argument is a special metaobject representing the name of the port and its arguments. If the port can be fired, this method returns a set of bindings of possible ways of firing.

`performBoundPort:` performs the bound port. This operation is called after the operation *testPort:* and its argument is a special metaobject representing the name of the port and the binding the port should be fired for.

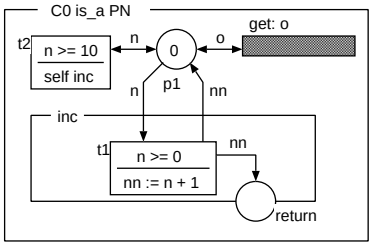
`step` performs one firable event. The event can be

- to fire a transition (creating the thread) including performing its first command
- to perform next command of the thread. The thread is finished along with the last command (it is taken as an atomic event). If the transition has no command, it is fired and finished as one atomic event.
- to finish the message processing. If some object is placed into the place `return` of the called process, it notifies the calling thread (the new event is created). When this event is performed, the thread acquires the return object and destroys called process.

5.1 Example: Simulation

Let us have simple example shown in the Figure 9. It creates new simulation world (`PNtalkWorld new`), new instance of the OOPN class `CO` (the class is shown in picture on the left) and puts this object into the world (`newIn:`). Because this operation returns a special proxy-metaobject (it will be explained in the section 6) we have to get the metaobject `PNObject` by calling `meta`. Then we get metaobjects stepwise: the object net (`componentID: 1`; each object net is always first process so that it has the number 1) and the place named `p1` (`placeNamed: 'p1'`). Now we put the object `11` into the place (`put: 11 mult: 1`) and have to test the changed object (we can test the whole simulation too as it is shown in out example—`world test`). Now we call the operation `step` of the world. This operation is called twice. The first calling causes the transition

`t2` is fired and the operation `inc` is evoked (the process is created). The second calling causes the transition `t1` is fired and the operation `inc` is finished along with the transition `t2` (transition firing and ending as well as process creating and finishing are understood as atomic operations from the one simulation step point of view). At the end, we can test if the place `p1` contains the object `12` (`contains: 12`).




```
world := PNtalkWorld new.
cls := rep componentNamed: 'C0'.
obj := cls newIn: world.
mobj := obj meta.
objnet := mobj componentID: 1.
place := objnet placeNamed: 'p1'.
place put: 11 mult: 1.
world test.
world step. world step.
place contains: 12. "=> true"
```

Fig. 9. The example of the OOPN simulation.

6 Communication mechanisms

As we already said, the formalism of OOPN and PNtalk are closely associated with Smalltalk environment. It implies that there can be native cooperation between OOPN and Smalltalk objects, so that it is possible to transparently access OOPN objects from Smalltalk and vice versa. It is also possible to use Smalltalk objects as the tokens in OOPN and to use PNtalk objects as Smalltalk objects.

To achieve the general communication between objects at different levels, the concept of *proxyobjects* is introduced. The proxyobject does not define the computational behavior of the receiver, but it ensures message passing in the requested way and it conforms the response to sent messages in accordance with the requirements of the sender. Thus, the proxyobject adapts the computational behavior of the receiver to the computational behavior of the sender.

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. However, the PNtalk proxyobjects have to implement additional properties necessary for the metaobject protocol.

The PNtalk 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 meta-level 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 class hierarchy of proxyobjects is shown in the Figure 10.

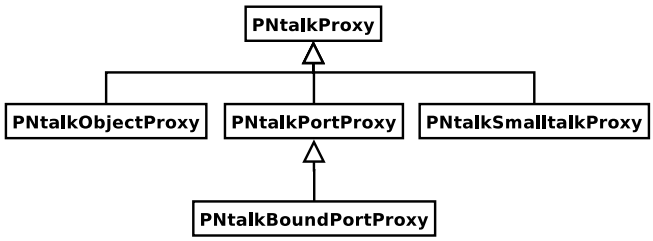


Fig. 10. The composition of the metaobjects *PNClass* and *PNObject*.

PNtalkObjectProxy It serves as a basic proxy-object to the PNtalk object represented by the metaobject *PNObject*. An object (potential sender of a message) always refers to another object (potential receiver) by means of a proxyobject. In spite of the domain level point of view, the actual sender is either the Smalltalk object or the PNtalk object. The receiver is always PNtalk object. Therefore, the metaobject *PNtalkObjectProxy* always refers to the metaobject *PNObject*. The list of selected meta-operations from the metaobject protocol follows:

`performDomainMessage:` performs a domain message. The metaobject *PNObject* should be always the sender, therefore this operation serves for message passing between PNtalk objects. It simply forwards the same message to the receiver (wrapped object).

`doesNotUnderstand:` operates a message unknown for the proxy-object. The message is a domain message of the receiver, the sender is not a metaobject but a Smalltalk object. It sends the caught message via the *performDomainMessage:* operation to the receiver and then waits for its result (when the process is being finished, it notifies this metaobject, see `step` in the *PNObject* metaobject protocol list). This message servers for communication from Smalltalk object to the PNtalk objects.

`asPort` returns a metaobject *PNtalkPortProxy* for the same receiver.

PNtalkPortProxy It servers for accessing PNtalk objects from Smalltalk objects at the level of ports. The list of selected meta-operations from the metaobject protocol follows:

`doesNotUnderstand:` operates a message unknown for the proxy-object. The message is a port calling of the receiver, the sender is not a metaobject but

a Smalltalk object. It sends the caught message via the *testPort:* operation to the receiver. It returns a metaobject *PNtalkBoundPortProxy* which represents the result of the port testing.

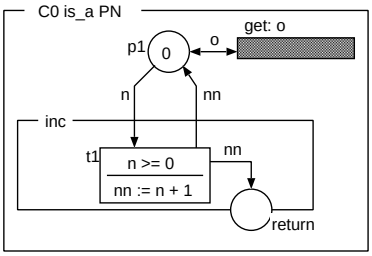
PNtalkBoundPortProxy It serves as an information storage for the bound port. It can be got only via the operation *testPort:* of the metaobject *PNtalkPortProxy*. The list of selected meta-operations from the metaobject protocol follows:

`ifTrue:` performs the specified block of commands if the port testing was successful (there is at least one possible binding)
`ifFalse:` performs the specified block of commands if the port testing was unsuccessful (there is no possible binding)
`collectBindings:` returns all bound variables for specified binding (the number). The operation returns it as an associated array (a.k.a. map or dictionary) of pairs (`name, value`).
`collectBindings` returns a collection of all bindings, for each binding it uses the operation `collectBindings:`.
`collectVariable:` returns a collection of all possible bindings of specified variable.
`variable:` returns a bound value of specified variable for the first binding.
`perform:` performs the port for the specified binding (the number).
`perform` performs the port for the first binding.

6.1 Example: Object Introspection

Let us have the simple example shown in the Figure 11. It supposes that there is a running default world and the PNtalk classes repository is accessible via the variable `rep`. First, we get the metaobject representing the PNtalk class `C0` (see the picture on the left). Then we create an instance of it—in fact, we get a proxy-object of `PNObjectProxy` to the metaobject of `PNObject`. Although the metaobject `PNObject` needs to receive domain message in a special way, we are able to send a message in the ordinary way via the proxy-object (see `obj inc`—the return value will be `1`).

The second part of this example shows how to use synchronous port with free (unbound) variables. We send the message in the same way but we use a special metaobject as the appropriate argument. In our example, we want to get a content of the place `p1` using the port `get:`. By calling `PNtalk variableNamed: #w`, we get the special metaobject representing the variable named `v`. Then we get a special metaobject for communication at the port level (`obj asPort`) and send the message for port named `get:` with free variable named `v`. The result is the metaobject *PNtalkBoundPortProxy* storing information about all possible bindings for variable `v` (it is just one object in our example). If the test is successful (see `port ifTrue:`) we can get a bound value of the variable `v` (see `port variable: #v`).




```
cls := rep componentNamed: 'C0'.
obj := cls new.
res := obj inc. "res is 1"
obj inc.
port := obj asPort get: (
      PNtalk variableNamed: #v).
port ifTrue: [
  res := port variable: #v.
  "res is 2"
].
```

Fig. 11. The example of the object introspection.

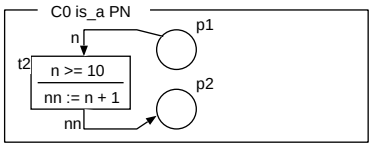
6.2 Example: Object Modification

Let us have the simple example shown in the Figure 12. It supposes that there is running default world and the PNtalk classes repository is accessible via the variable `rep`. First, we get the metaobject representing the PNtalk class `C0` (see the picture on the left). Then we create an instance of it. We can compile new elements into the objects: two synchronous ports `put:` and `get:` (see `obj meta compile: ...`). The resulted object net is shown in the picture on the right. Now, we can communicate with the object using these ports. First, we put a number `20` into the place `p1` (see a sequence of `obj as Port put: 20` and `p perform`). The transition `t1` will be fired because the world is running and the transition becomes firable immediately the port is performed. Second, we put a number `30` into the place `p1` by the same way.

Now, we can get the content of the place `p2` using the synchronous port `get:` (it is the same principle as described in the Example 11). We can get a collection of all bindings of the variable `v` (see `p collectVariable: #v`; the collection will contain numbers `21` and `31`). When we perform this port for the first binding (`v==21`), the number `21` will be removed from the place `p2`.

7 Conclusion

We have presented the basis of the PNtalk system architecture and have demonstrated its features by simple examples. The goal of the PNtalk project is not only to make a tool intended for modeling and simulation but also to make tool allowing the developed model to be integrated into 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 the model specification, to experiment with new paradigms, or to allow both interactive and automatic evolution of the models. For instance, the reflection was used for merging OOPN and DEVS formalisms.




```
cls := rep componentNamed: 'C0'.
obj := cls new.
obj meta compile: 'sync put: o
  postcond p1(o)'.
obj meta compile: 'sync get: o
  precond p2(o)'.
p := obj asPort put: 20.
p ifTrue: [ p perform ].
p := obj asPort put: 30.
p ifTrue: [ p perform ].
p := obj asPort get: (
      PNtalk variableNamed: #v).
p ifTrue: [
  res := p collectVariable: #v.
  "res is a collection (21 31)"
  res perform: 1.
  "the place p1 contains 31"
].
```

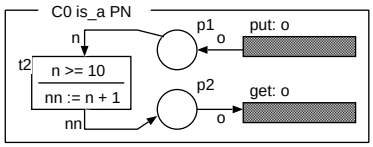


Fig. 12. The example of the object modification.

One of possible application domains is artificial intelligence, especially the area of intelligent multi-agent systems. One of our experimental applications is PNagent [13]. It is a framework for rational agents development. It uses fragments of plans specified by Petri nets and the whole framework is implemented using OOPN in PNtalk. Consequently, it is possible to continually develop both the agents and the agent framework using the same language featuring both visual representation and formal basis. We have experimented with simple case studies [11, 18] in the field of software engineering, too.

Acknowledgments. This work was supported by the Czech Grant Agency under the contracts GA102/07/0322, GP102/07/P306, and Ministry of Education, Youth and Sports under the contract MSM 0021630528.

References

1. The TUNES Project for a Free Reflective Computing System. http://tunes.org, 2006.
2. F. J. Barros. Modeling formalisms for dynamic structure systems. In *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 1997.
3. J. Briot. 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*. Herms Science, 1999.
4. M. Ceska, V. Janousek, and T. Vojnar. PNtalk — a computerized tool for object oriented petri nets modelling. In *EUROCAST*, pages 591–610, 1997.
5. A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, 1989.
6. V. Janousek and E. Kironsky. Reflective Framework for Interactive Modeling and Simulation of Intelligent Systems. In *Proceedings of Nineteenth International Con-*

ference on Systems Engineering 19-21 August 2008 Las Vegas, Nevada, Los Alamitos, US, pages 480–485. IEEE Computer Society, 2008.
7. V. Janoušek. PNtalk: Object Orientation in Petri nets. In *Proc. of European Simulation Multiconference ESM'95*. Prague, CZ, 1995.
8. V. Janoušek. *Modeling objects by Petri nets (in czech)*. PhD thesis, 1998.
9. V. Janoušek and R. Kočí. Towards an Open Implementation of the PNtalk System. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation*. EUROSIM-FRANCOSIM-ARGESIM, Paris, FR, 2004.
10. V. Janoušek and R. Kočí. Towards Model-Based Design with PNtalk. In *Proceedings of the International Workshop MOSMIC'2005*. Faculty of management science and Informatics of Zilina University, SK, 2005.
11. V. Janoušek and R. Kočí. Simulation and design of systems with object oriented petri nets. In *Proceedings of the 6th EUROSIM Congress on Modelling and Simulation*, page 9. ARGE Simulation News, 2007.
12. G. Kiczales, J. Lamping, A. Menhdhekar, Ch.Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241 of *1241*. Springer-Verlag, 1997.
13. R. Kočí, Z. Mazal, F. Zbořil, and V. Janoušek. Modeling Deliberative Agents Using Object Oriented Petri Nets. In *Proceedings of the 7th ISDA*, pages 15–20. IEEE Computer Society, 2007.
14. C. A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *Proceedings of the Application and Theory of Petri Nets*, volume 935. Spinger-Verlag, 1995.
15. Ch. Lakos. Towards a Reflective Implementation of Object Petri Nets. In *Proceedings of TOOLS Pacific*. Melbourne, Australia, 1996.
16. P. Maes. Computational reflection. Technical report, Artifical Inteligence Laboratory, Vrije Universiteit Brusel, 1987.
17. D. Moldt. OOA and Petri Nets for System Specification. In *Application and Theory of Petri Nets; Lecture Notes in Computer Science*. Italy, 1995.
18. R. Kočí and V. Janoušek. System Design with Object Oriented Petri Nets Formalism. In *The Third International Conference on Software Engineering Advances Proceedings ICSEA 2008*, pages 421–426. IEEE Computer Society, 2008.
19. C. Sibertin-Blanc. Cooperative Nets. In *Proceedings of Application and Theory of Petri Nets*, volume 815. Springer-Verlag, 1994.
20. A. M. Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach. In *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, volume 11, 2001.
21. R. Valk. 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*, volume 120. Springer-Verlag, 1998.
22. Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27. ACM Press, New York, 1992.
23. B. Zeigler, T. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., London, 2000.