Hierarchical and Time Extensions of Pure Object Oriented Petri Nets*

Tomáš Vojnar

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

Abstract: *The aim of this article is to suggest a suitable way of combining pure object oriented Petri nets (OOPNs) connected to the language PNtalk with hierarchical and time constructions. Incorporating substitution transitions and fusion places into OOPNs is discussed, together with its advantages in the area of modelling. Further, the concepts of delay and firing time are introduced into PNtalk here. Terms like age memory policy, enabling memory policy, or single and multiple server are adapted for the environment of OOPNs. However, not only stochastic time extensions are taken into account, but also time extensions suitable for evaluating real-time properties are covered.*

Key words: Petri nets, object orientation, time extensions of Petri nets, hierarchy in Petri nets, simulation and modelling.

1 Introduction

Petri nets combine a graphical way of describing models with a possibility of applying powerful simulation and analysis techniques. Object oriented Petri nets were designed to bring the object oriented structuring paradigm based on classes and methods into unstructured Petri nets. In this way also all general advantages of object orientation, e.g. encapsulation, inheritance, and polymorphism, have been brought into the area of Petri nets.

However, it seems that there are situations where the flexibility of communication by means of message sending is unnecessary and modelling based on classical hierarchical constructs, namely substitution transitions, can lead to less complex and more readable models. This idea is discussed in more details in the first part of this article. And since we eventually conclude that it is advantageous to support substitution transitions also in the framework of OOPNs, a suitable way of their incorporating is suggested.

In the second part of the article there are discussed various time extensions of Petri nets, which are necessary for validating not only logical correctness of systems modelled by Petri nets but also their performance or real-time characteristics. We again suggest a way of incorporating these extensions into the pure OOPN formalism. For this purpose we use special pseudo-methods and thus we do not have to extend the present syntax of OOPNs.

*A shortened version of this article was published in the proceedings of ASIS'97, Krnov, Czech Republic, MARQ Ostrava, 1997.

2 A Brief Description of the OOPN Formalism

This section provides a very brief description of the OOPN formalism just to make the article self-contained. A more detailed description can be found e.g. in [12].

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 can access the corresponding object net (places of the 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 inheritance of object nets, together with sets of method nets.

There exist also special method nets which play the role of constructors. Further, predicates, which have the form of transitions with associated message patterns, can be used for testing objects states.

Particular nets consist of places and transitions interconnected by arcs. Every place has an initial marking and an initial action, every transition has a guard and an action. Arcs are inscribed by arc expressions which specify what tokens will be removed or added with respect to a concrete binding. From the point of view of transitions, there are input, output, and test (dialog) arcs.

Tokens in a running OOPN represent objects. 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 being 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 sending and object creations are specified as actions attached to transitions. A transition can be fired for a particular binding, if there are enough tokens in its input places with respect to the values of arc expressions in the given binding and if the transition guard evaluates to `true`. Then tokens from input places are removed, the transition action is performed, and as soon as this is finished, the output marking is produced.

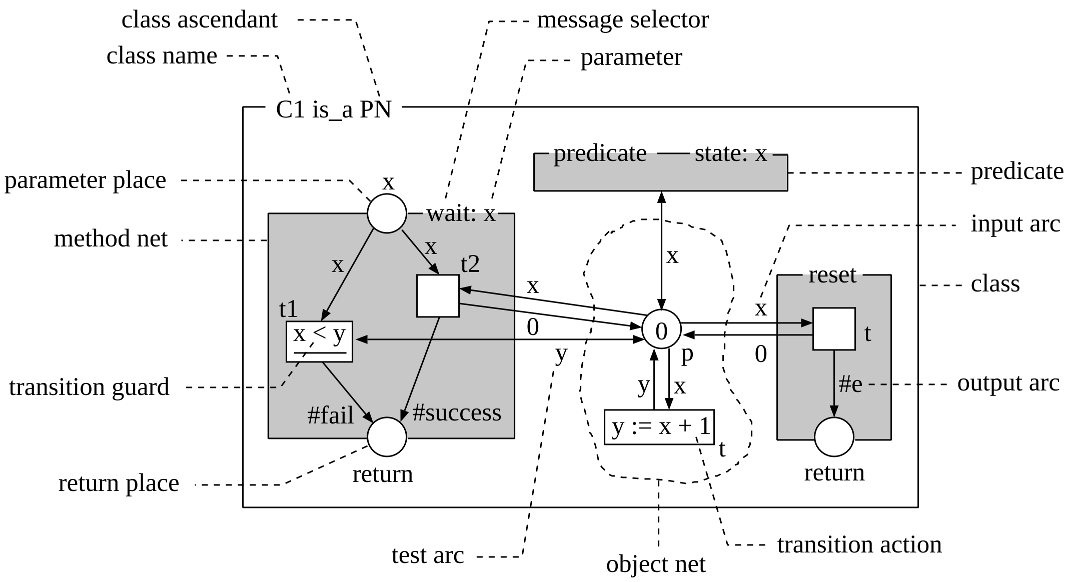

Figure 1: A simple class demonstrating the notion of the OOPN formalism.

3 Classical Hierarchical Constructs in the OOPN Framework

In this section we will compare the notion of substitution transitions[1], common in the area of classical hierarchical Petri nets [1, 2], with the message sending mechanism[2], which is typical for OOPNs. We will advocate usefulness of substitution transitions even in the object oriented environment and we will adopt them for this environment. Then we will have a brief look at another classical construct – fusion places, too.

It is obvious that the message sending mechanism is more general than the usage of substitution transitions – it allows us to write recursive definitions and to communicate with different receivers at different times. Message sending as a part of the object oriented paradigm also brings us the possibility of polymorphism. However, when communicating objects are well known and a communication channel between them is fixed, the notion of substitution transitions can yield a briefer and better readable description, which is demonstrated by a simple example in Fig. 2. In this example we model a situation when we want to have a fixed channel which immediately transfers everything from place `s` into place `p` of the component of type `C2`. We see that it is enough to specify `s` as a socket, `p` as a port, and `C2` as a substitution transition, if we can use classical structuring mechanisms[3]. Whereas in the case of message sending, it is necessary to create a dedicated method for storing objects into `p`, a special place containing the component of type `C2`, and as a consequence the whole description is obviously less readable. And the situation gets even more complicated when we want to be able to communicate in the same way also from `p` to `s` (a two-way channel). The same is true when we want to create a place serving like a shared memory for several components (see Fig. 3), which are again fixed.

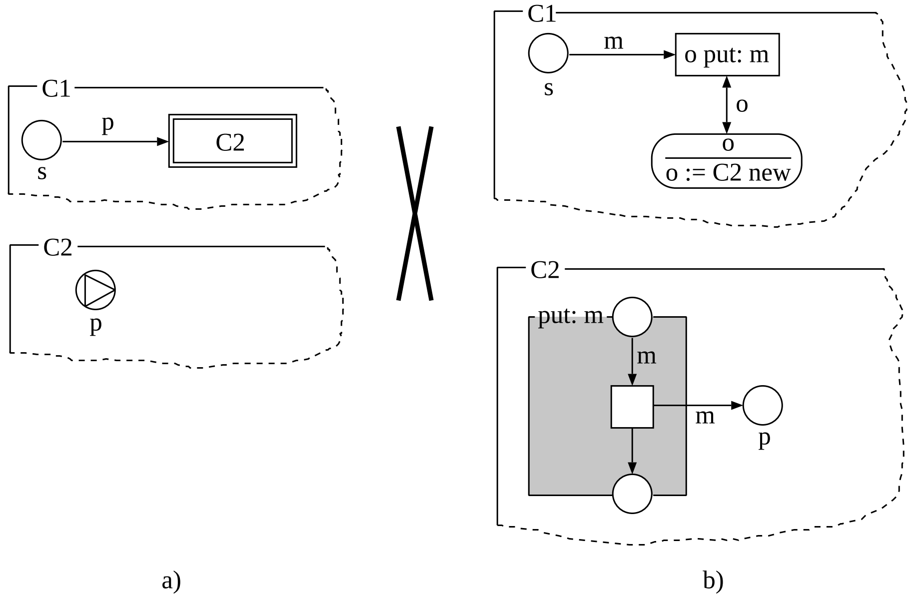

Figure 2: A comparison of modelling by (a) substitution transitions and (b) message sending.

[1]Substitution transitions represent subnets by which they are replaced during the initialisation phase. This is done by fusing corresponding port and socket places. These transitions play the role of macros known from classical programming languages.

[2]A message is sent to an object from another object, the corresponding method is invoked as a response to this message, some computation is performed, and then the result is returned to the message sender.

[3]The syntax of using substitution transitions is different from the one introduced in [1, 2] and will be explained later.

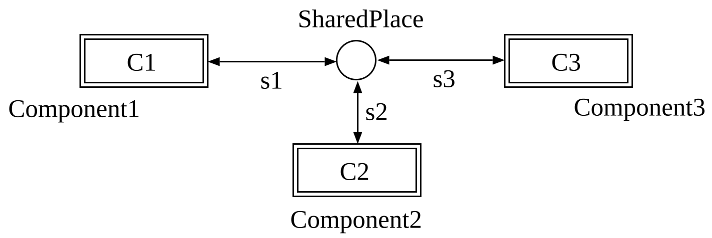

Figure 3: A place shared by several components modelled by substitution transitions.

The above described cases in which it is advantageous to use substitution transitions are quite frequent in practice, e.g. when we model some electronics devices (chips) or flexible manufacturing systems, their components are fixed (there usually do not arise or disappear machines) and the components are interconnected unalterably (there do not arise or disappear conveyor belts or manipulators round a machine). The usefulness of this construction was also proved by many successful applications of the Design/CPN tool [9], which is based on it.

Let us return back to the message sending for a while. Its advantages can be found in another areas of modelling and it also leads to a different modelling style. For example, it is suitable to use message sending when modelling systems with active objects which can meet, communicate for some time, and then separate again. This is typical especially for systems comprising people, see e.g. the model of a society of patients and doctors in [12], where a patient can ask a doctor to help him, the doctor examines him, announces him a cure, and then the patient leaves the doctor. Further, we cannot forget to stress the possibility of modelling recursion by means of message sending. Finally, message sending allows us to communicate with the appropriate components (objects, pages) from more than one position in the model, i.e. from different nets, objects, or – in the terminology of Design/CPN – pages. This problem is solved in traditional hierarchical Petri nets by introducing fusion places. However, it is difficult to use fusion places to model a situation when we want to communicate with components of the same type from several places of a model but with each component (or group of components) separately.

With respect to the previous paragraph, it appears it would be good to have a possibility to communicate with objects introduced via substitution transitions not only by socket and port nodes but also by message sending. We can e.g. let neighbouring machines exchange products by sockets and ports and at the same time these machines can be controlled from some control objects by message sending.

Now we can summarise that it seems that substitution transitions can be useful also in the framework of OOPNs and we can turn our attention to the way how to incorporate them into this formalism in a consistent way. Since nearly everything in an OOPN is object, we will want so that substitution transitions will also give rise to objects connected to superior objects by means of ports and sockets. These objects should be instances of regular classes, which will allow us to apply mechanisms of inheritance, message sending, etc. Nevertheless, in order not to violate the encapsulation mechanism, it is necessary so that it will be clear which places can be used as ports.

We suggest the syntax of using substitution transitions sketched in Fig. 4. Substitution transitions are drawn as transitions with double border line and they are inscribed by the name of a class whose objects will be connected to the appropriate sockets. Sockets are common places, whereas ports are special places which are visible from outside of classes. There are three types of ports: input, output, and input/output ports. Input ports can be adjacent to outgoing arcs only and, what is more, they can be referenced only from incoming arcs of substitution transitions. The opposite is true for output ports and input/output ports are a mixture of both previous types of ports. Substitution transitions are connected to sockets by arcs inscribed by corresponding ports. We restrict the possibility of using substitution transitions to object nets only.

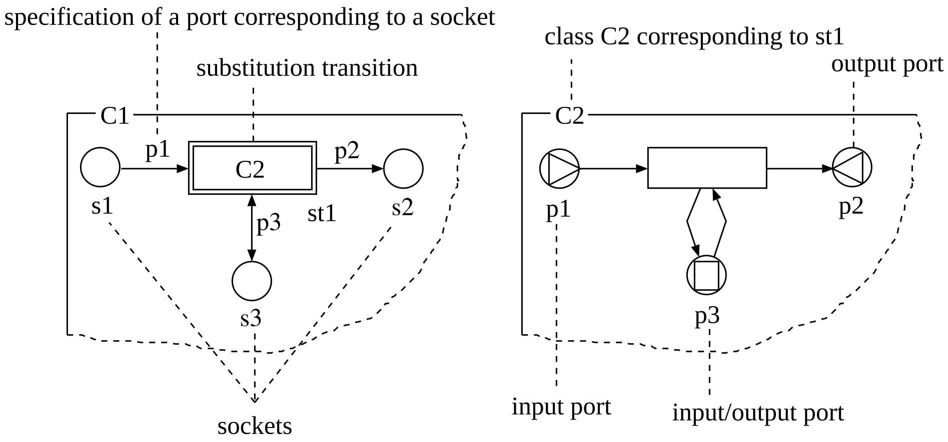

Figure 4: The syntax of using substitution transitions.

The semantics of the above introduced construct is as follows. Every time a new instance of a net comprising substitution transitions is created, new instances of the classes specified in the substitution transitions are created and connected to their environment via ports and sockets. This connection ensures that as soon as a token appears in a socket, this token can be consumed from the corresponding input port. Similarly, as soon as a token appears in an output port, it can be consumed from the appropriate socket. Finally, the semantics of input/output ports leads to a full fusion of corresponding ports and sockets.

The semantics of substitution transitions can be specified also be means of unfolding into pure OOPNs (see Fig. 5). Thus it is not necessary to modify the formal definition of OOPNs to be able to work with substitution transitions. However, it is not so good to use this unfolding during simulation or analysis because of an unnecessary contribution of this unfolding to the state explosion problem.

The way we have introduced substitution transitions into the OOPN formalism allows us to use the already mentioned mixed communication. We can realize it by creating an input/output port with `self` as its initial marking and by connecting this port to a suitable socket. From this socket we can obtain a reference to the object which is launched by the substitution transition at the time of instantiation of the net in which this transition is situated. Then, the obtained reference can be used for message sending in a common way. The above described mechanism is illustrated in Fig. 6. It is also obvious that this mechanism can be optimised by introducing a special class containing port `Self` and then using this class as an ancestor of all classes which we want to use for mixed communication.

At the end of our discussion of substitution transitions in the framework of OOPNs we should note that this mechanism is not an abstraction in the sense how it was introduced in [10]. This is because when we look at a substitution transition we cannot say nearly anything about its behaviour unless we know something about the referenced class. On the other hand, there is some analogy between superplaces and method nets and between supertransitions and transitions with method calls. This similarity has already been mentioned in [10] in the context of nets used in the tool called SYROCO [11]. However, it is still necessary to solve questions

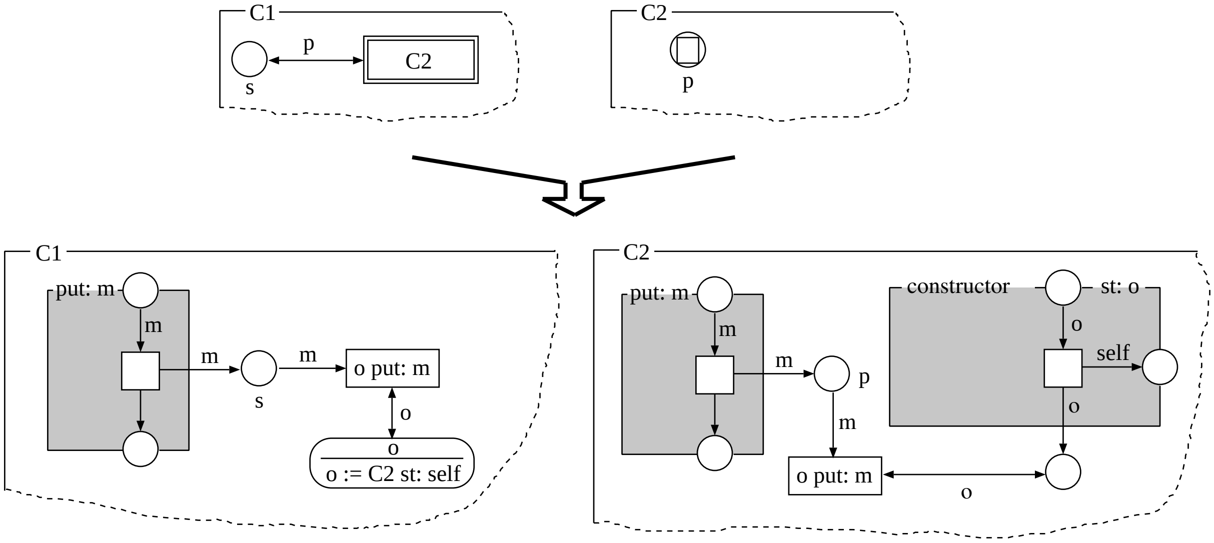

Figure 5: An example of an unfolded substitution transition with an input/output port.

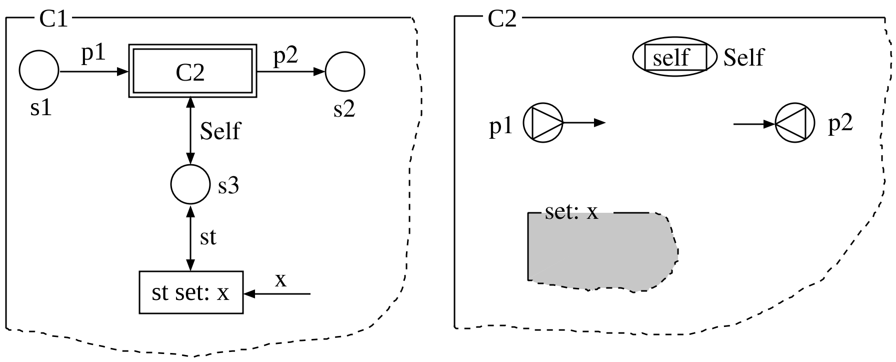

Figure 6: An example of a mixed communication with an object.

connected to the dynamic instantiation of nets and to the possibility of recursive calls.

After having discussed substitution transitions, we will shortly mention also another structuring mechanism of classical hierarchical Petri nets, namely the mechanism of fusion places. Up to now we allow using fusion places only within particular net instances, where they play a role of some kind of syntactic sugar, which can be used for avoiding crossing of too many arcs. Usefulness of fusions within all instances of a given net or even with a broader scope remains an open question, together with its graphical representation and its relation to the object orientation.

4 Time Extensions of OOPNs

If we want to use Petri net based models not only for evaluating logical correctness but also to assess performance or real-time characteristics of systems, it is necessary to introduce some concept of time into the kind of Petri nets we intend to work with. And in fact, there have already been presented very many different time extensions of Petri nets – see e.g. [3, 4, 5, 6, 7] and many others. These approaches differ e.g. in the kind of Petri net elements they use for introducing time – we can find time associated with places, transitions, tokens, or arcs. Further, this approaches differ in their expressive power and also in the class of models they are intended for. There are two main classes of Petri nets augmented with time concepts – stochastic Petri nets and time or timed Petri nets. Stochastic Petri nets specify duration of actions by means of probability distributions and they are usually used for evaluating performance characteristics of systems. Their analysis is based on transformations to various kinds of Markov chains. On the other hand, time or timed Petri nets work with constant length of actions or at most with minimum and maximum ranges of duration of these actions. They can be used for assessing performance as well, but it is more typical to use them when dealing with real-time characteristics. They are usually used with some kind of temporal logic for expressing desired properties and their analysis is based on constructing timed reachability graphs.

We will connect our time extension of OOPNs to transitions and we will exploit both commonly used concepts of timing based on transitions – delay and firing times. The delay time concept can be used for specifying how long some tokens corresponding to a given binding must be present in input places of a given transition before this transition can be fired with the given binding. When applying the firing time construct, we can say how long it will take before output tokens will be produced after removing input tokens of a given transition for a given binding. We should note that the delay time concept is more general than the firing time one – everything what can be expressed by firing times can be described also by delay times but not vice versa. This is because it is impossible to model interrupts of timed actions by means of firing times. However, we will work with both concepts, because situations when it is possible to use firing times are quite frequent and if had to use delay times, we would have to add many unnecessary places and transitions. Further, it is also necessary to point out that delay times are defined for particular bindings and a time which elapsed during waiting for the end of a delay of one particular binding cannot be used for covering a part of the delay of a different binding of the same transition. This is illustrated by a simple unfolding of a part of an OOPN model into a GSPN fragment in Fig. 7.

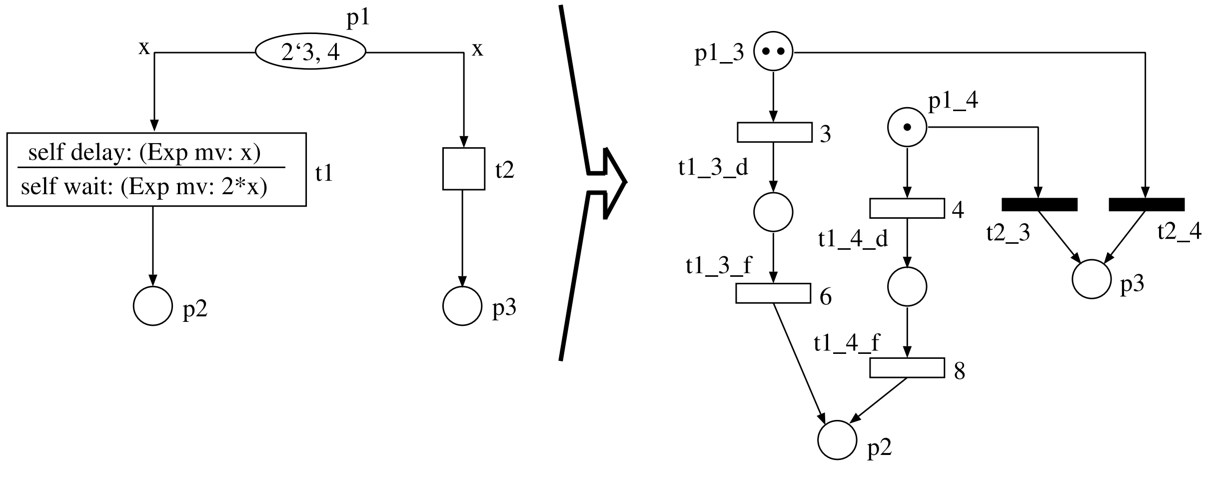

Figure 7: An unfolding of an OOPN fragment to a GSPN.

By introducing the delay and firing time concepts we have increased the potential amount of state information which can be bound to transitions. This is because we have to store information about bindings for which the the delay or firing countdown is running and we always have to know when this countdown will reach zero.

Now it is the right time to say something about syntax of our extensions. We will base it on the well known syntax of message sending. Thus we will be specifying delay times by formally sending a new pseudo-predicate message `delay:` to `self` in guards. Similarly, we will introduce a pseudo-method `wait:` applicable in actions by means of which we will be able to express firing times. This attitude is motivated by the fact that delay times in some way restrict firability of transitions, which is similar to application of predicates in guards, and firing times make transition actions non-atomic, which is typical also for methods called from actions. However, this syntax has not been completely fixed yet and may be, that some special syntactic and graphical representation will be introduced according to experience gained from planned case studies.

It is interesting that for the sake of simulation, `wait:` can be easily implemented as an ordinary method of the class `PN`, which is an ancestor of all OOPN classes. Then it is possible to perform timed simulations without any other significant modifications of the simulator. Unfortunately, this is not the case of `delay:`, where there are problems with identifying transitions and bindings for which the delay is specified (an implicit parameter would be necessary) and further with multiple calls of `delay:` during a single delay period (at the beginning a length of the period must be specified and then it is only tested for expiring). This is also complicated by a possible multiple enabling of some bindings and by the possibility of interrupts of timed actions. Therefore it is better to recognise calls of `delay:` already during the syntactic analysis phase and to treat them in a special way rather than to try to implement this pseudo-predicate as a boolean method. In the case of analysis, it will be necessary to recognize calls of both `delay:` and `wait:` and to process them in a special way, different from calls of ordinary methods and predicates.

We have introduced a pseudo-method and a pseudo-predicate allowing us to specify delay and firing times by means of their parameters. However, we have not said anything about the form of these parameters yet. Neither we have specified whether we work with stochastic or "real-time" timing. Let us to fill this gap now. We will not explicitly restrict ourselves to either stochastic or real-time modelling. Instead, we will allow modellers to create both kinds of models and only at the beginning of simulation or analysis we will have to find out what type of model we have to deal with. This decision will be based just on the used form of `delay:` and `wait:` parameters. In the case of stochastic models, the modellers will be able to use pseudo-classes representing various probability distributions, e.g. `Exp`, `Uni` etc., and pseudo-methods specifying concrete parameters of these distributions, e.g. `mv:` corresponding to the arithmetic mean. In the case of real-time models, it will only be possible to specify delays by single values, possibly derived from transition variables, or by means of the pseudo-class `Lim` and the pseudo-method `min:max:`, which will allow the modellers to specify minimum and maximum duration times of actions. For the sake of simulation, it will be again possible to implement the recently introduced pseudo-classes and pseudo-methods as regular classes and methods. However, it is necessary to note that using simulation for validating real-time models is a rather unusual and desperate choice.

To allow modellers to create full-valuable stochastic models we have to introduce some further concepts, namely various kinds of memory policies and the concepts of finite and infinite servers. From among the various memory policies we will choose two – age memory policy and enabling memory policy. The first one states that, after an action has been interrupted, as soon as it can go on, only its not yet performed part has to be done. The latter one, then, restarts the action from its beginning. Infinite server semantics means that infinitely many timed actions can be running at the same time and finite server semantics restricts the number of concurrent actions to some finite number, typically one.

We will define infinite server semantics and enabling memory policy to be default. To make it possible for a modeller to explicitly change the memory policy to the age one, we will define another pseudo-method – `ageMemory` – applicable in guards only. This is because it is sensible to use different memory policies only with the delay time concept. When using the age memory policy, modellers must be aware of the fact that there are different "memories" storing already passed parts of delay times for different bindings, which is also consistent with the unfolding in Fig. 7. As for the finite servers, it is easy to introduce another pseudo-method, e.g. `finiteServer:`, but than it is not so easy to define its semantics in the context of tokens with colours. It would not be so difficult, if we joint capacities with particular bindings, i.e. a transition could be enabled only finitely many times for a given binding without any respect to its enabling for different bindings. However, if we want to restrict the capacity of a server within all possible bindings, the semantics gets a bit more tricky. Although we can define it in such a way that only several countdowns can be running for a given a transition at a given time and a new one can be started or unblocked only when some active timer reaches zero or is blocked, it is not so clear how to describe this in the corresponding place/transition stochastic net (e.g. in the one in Fig. 7). A solution of this problem remains a question of the future research.

After having introduced some special concepts for stochastic models, we have to add one also for real-time models. It is the possibility of distinguishing transitions with weak and strong time semantics. The weak time semantics says that a transition *can* fire within the given bounds. On the other hand, the strong semantics states that a transition *must* fire within the given bounds. Let us define strong semantics as default and introduce the pseudo-method `weakTiming` applicable in guards and allowing us to use weak time semantics, too.

At the end of this section we should mention that there are possible many other time extensions. Currently especially fuzzy timing is quite up-to-date – see e.g. [7]. Further, it would also be possible to allow using local timers in particular objects or nets, which is considered to be useful e.g. in [8] and it is also applied in the tool SYROCO [11] which works with a formalism quite similar to the formalism we use in this article.

5 Conclusions

We have enriched the pure OOPN formalism by substitution transitions, very well known in the area of classical hierarchical Petri nets, and by several time extensions. It seems that substitution transitions could be successfully used for modelling some classes of systems, e.g. flexible manufacturing systems. This should be verified on some suitable non-trivial case studies, which could also help us to improve the graphical representation of substitution transitions and the way they are combined with the message sending mechanism. However, a necessary condition for any case study is incorporating substitution transitions not only into the OOPN formalism but also into the tool based on it, i.e. into the PNtalk system. A PNtalk system modification and subsequent case studies are needed also for verifying the way we have introduced time into OOPNs. Finally, it is necessary to adapt or work out analysis techniques both for stochastic and real-time models based on OOPNs.

Acknowledgment

This work was supported by the grant of the Czech Grant Agency GAČR No. 102/96/0986, by the Universities Development Fund of the Czech Ministry for Education under the contract F1: 618/1997, and finally by the grant of the Fund for Sciences and Arts of the Technical University of Brno under the contract FU470006.

The author would also like to express his special thanks and admiration to all the readers who read this article full of doubts and open questions.

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

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

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

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

[7] Murata, T.: *Temporal Uncertainty and Fuzzy-Timing High-Level Petri Nets.* In: Proceedings of ICATPN'96, Osaka, Japan, Lecture Notes in Computer Science 1091, Springer-Verlag, 1996.

[8] Blümel, P., Kindler, E.: *Simulation of Antagonist Systems - First Steps.* In: Proceedings of MOSIS'97, Hradec nad Moravicí, Czech Republic, MARQ Ostrava, 1997.

[9] Jensen, K.: *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use, Volume 3.* Springer-Verlag, 1997.

[10] Lakos, C.: *On the Abstraction of Coloured Petri Nets.* In: Proceedings of ICATPN'97, Toulouse, France, Lecture Notes in Computer Science 1248, Springer-Verlag, 1997.

[11] Sibertin-Blanc, C.: *SYROCO: A Distributed System of Cooperative Objects.* In: Tool presentations within ICATPN'97, Toulouse, France, 1997.

[12] Češka, M., Janoušek, V., Vojnar, T.: *PNtalk – A Computerized Tool for Object Oriented Petri Nets Modelling.* In: Proceedings of EUROCAST'97, Las Palmas de Gran Canaria, Canary Islands, Spain, to be published in Lecture Notes in Computer Science, Springer-Verlag, 1997.