# Type Analysis in Object-Oriented Petri Nets

Bohuslav Křena*, Tomáš Vojnar *
{krena,vojnar}@dcse.fee.vutbr.cz

**Abstract:** The paper considers the problem of an automatic type analysis in the context of the object-oriented Petri nets (OOPNs) associated with the PNtalk language. We describe the skeleton of one of the possible approaches to deriving the types of tokens that may get into particular places of OOPN-based models. We briefly discuss advantages and disadvantages and also possible alternatives to the described approach.

**Keywords:** object-oriented Petri nets, type analysis, formal techniques.

## 1 Introduction

This paper presents one of the research activities related to the object-oriented Petri nets (OOPNs) associated with the language and tool called PNtalk [3], which have been developed to support modelling, investigating, and prototyping concurrent object-oriented software systems. PNtalk supports intuitive modelling of all the key features of these systems, including object-orientedness, message sending, parallelism, and synchronisation. There has already been implemented a tool for simulating systems described by OOPNs [1]. Currently, we are working both on an implementation of a tool which should allow us to run OOPN-based prototypes in a truly distributed way as well as on methods for formally analysing and verifying properties of systems modelled by OOPNs [2].

PNtalk OOPNs—unlike most of the common dialects of high-level Petri nets [5]—are not strongly typed, which means that modellers do not have to explicitly declare the types (i.e. colour sets) of OOPN places, the types of the variables used in OOPN inscriptions, and so on. This feature of PNtalk is related to the fact that it has been inspired by Smalltalk and that it is intended to be used in the area of prototyping. However, as mentioned below, there are situations in which it may be useful to know at least something about the types associated with particular OOPN places, inscription variables, or other kinds of OOPN elements. Therefore this paper considers the possibility of an automatic type analysis for the context of PNtalk OOPNs.

The type analysis can be understood as a special means of debugging OOPN-based models: Modellers usually have some intuition about what types of tokens should get into particular OOPN places and if the results of the type analysis are different, there may be a fault in the appropriate model. The information derived from the type analysis can then also be used as a part of the documentation of a model. Moreover, as most Petri net dialects as well as various other formalisms seem to be strongly typed, the results obtained from the type analysis can further be useful when translating PNtalk OOPNs into models described by some other modelling languages. The possibility of such a translation may be interesting from the point of view of re-using the tools already developed for working with various kinds of models. Finally, the results which may be obtained from the type analysis may be used to optimise the internal representation of OOPNs and the efficiency of running OOPN-based models and generating their state spaces.

* Department of Computer Science and Engineering, TU Brno, Božetěchova 2, CZ-612 66 Brno

In the following, we firstly present the basic principles of OOPNs. Subsequently, we describe the skeleton of one of the possible approaches to the automatic type analysis in the context of OOPNs. Next, we discuss advantages and disadvantages and also possible alternatives of the described approach.

## 2 Object-Oriented Petri Nets

The OOPN formalism is characterised by a Smalltalk-based object-orientation enriched with concurrency and polymorphic transition execution, which allows message sending, waiting for and accepting responses, creating new objects, and performing primitive computations [3]. An example illustrating the notation of OOPNs is shown in Fig. 1.

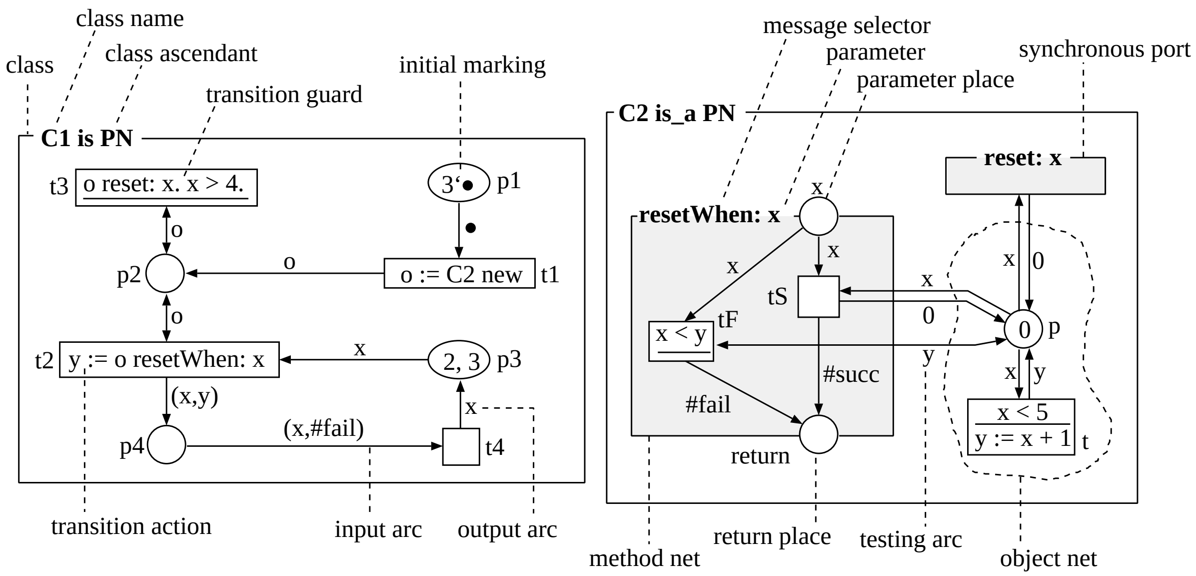



**Fig. 1:** Two simple classes demonstrating the notion of the OOPN formalism

The main principles of the structure and behaviour of OOPNs are explained in the following. A deeper introduction to the OOPN formalism can be found in [1] and the formal definition of OOPNs in [3].

### 2.1 The Structure of OOPNs

An *object-oriented Petri net* is defined on a collection of elements comprising constants, variables, net elements (i.e. places and transitions), class elements (i.e. object nets, method nets, synchronous ports, and message selectors), classes, object identifiers, and method net instance identifiers. An OOPN has its initial class and initial object identifier, as well. The so-called universe of an OOPN contains (nested) tuples of constants, classes, and object identifiers.

*Object nets* consist of places and transitions. Each place has some (possibly empty) initial marking. Each transition has conditions and preconditions (i.e. inscribed testing and input arcs), a guard, an action, and postconditions (i.e. inscribed output arcs). *Method nets* are similar to object nets but, additionally, each of them has a set of parameter places and a return place. Method nets can access places of the appropriate object nets, which allows running methods to modify the states of the objects which they are running in.

*Synchronous ports* are special transitions which cannot fire alone but only dynamically fused to some regular transitions. These transitions "activate" the ports from their guards via message sending. Every synchronous port embodies a set of conditions, preconditions, and postconditions over places of the appropriate object net, and further a guard, and a set of parameters. Parameters of an activated port $s$ can be bound to constants or unified with variables defined on the level of the transition or port that activated $s$.

A *class* is given by its object net, its sets of method nets and synchronous ports, and a set of message selectors corresponding to its methods and ports. Object nets describe what data particular objects encapsulate and what activity they exhibit on their own. Method nets specify how objects asynchronously respond to received messages. Synchronous ports allow to remotely test and change states of objects in an atomic way.

### 2.2 The Dynamic Behaviour of OOPNs

A state of an OOPN can be encoded as a *marking*, which can be structured into a system of objects. Thus the dynamic behaviour of OOPNs corresponds to an evolution of a system of objects. An *object* of a certain class $c$ is a system of net instances that contains exactly one instance of the object net of $c$ and a set of currently running instances of method nets from $c$. Every *net instance* entails its identifier and a marking of its places and transitions. A *marking of a place* is a multiset of elements of the universe. A *transition marking* is a set of records about method net instances invoked from the appropriate transition.

For a given OOPN, its *initial marking* corresponds to a single, initially marked object net instance from the initial class. A change of a marking of an OOPN is a result of an occurrence of some *event*. Such an OOPN event is given by (1) its type, (2) the identifier of the net instance it takes place in, (3) the transition it is statically represented by, and (4) the binding tree containing the bindings of the variables used on the level of the involved transition as well as within all the synchronous ports (possibly indirectly) activated from that transition. There are four kinds of events according to the way of evaluating the action of the appropriate transition: `A` – an atomic action involving trivial computations only, `N` – a new object instantiation via the message `new`, `F` – an instantiation of a Petri-net described method, and `J` – terminating a method net instance.

## 3 Type Analysis in OOPNs

### 3.1 Different Approaches to Type Analysis in OOPNs

There can be identified two extreme approaches to deriving the types of tokens which may get into particular places of an OOPN. Firstly, we may immediately assign every OOPN place the type corresponding to the union of all the types which appear in the universe of the given OOPN. On the other hand, we may generate and explore the full state space of the OOPN (if the state space is not infinite), which allows us to precisely find out what types of tokens may get into particular places (with or without mixing the marking of different net instances). It is obvious that the first approach may be extremely inaccurate, while the second one quite expensive [6] (and, without some modifications, inapplicable in the case of infinite state OOPNs).

In the rest of the paper, we introduce two specialised approaches to analysing types in OOPNs, namely the so-called *static* and *dynamic OOPN type analysis*. The accuracy and the computational costs of these methods are normally between the above mentioned extremes. Both of the methods somehow approximate the behaviour of OOPNs and can handle infinite state OOPNs, as well.

Most of the principles of the static and dynamic OOPN type analysis are quite similar. In this paper, we informally present the kernel of the algorithm for the static type analysis, which works purely on the level of particular nets and entirely ignores that there can be created multiple, mutually independent instances of them. The dynamic type analysis (briefly mentioned in the end of the section) attempts to distinguish particular net instances to some degree. Therefore the dynamic type analysis is more accurate, but it is more complex and has higher time and space requirements.

### 3.2 The Basics of the OOPN Static and Dynamic Type Analysis

In order to decrease the time and space requirements of the type analysis and to ensure its termination, the static and dynamic OOPN type analysis methods suitably approximate the behaviour of OOPNs. The price which we pay for this is that the sets of the types of tokens about which we are informed that they can get into particular places can be bigger than necessary. However, this is a safe approximation, and the obtained results can still be quite useful.

First of all, the static and dynamic OOPN type analysis methods work with symbolically represented markings and events which do not contain concrete constants and unambiguous identifiers of Petri net-based objects, but only the corresponding types (extended with some auxiliary data in the case of the dynamic type analysis). Accordingly, we do not perform classical trivial computations in guards of transitions and ports or in actions of transitions. Instead, we only derive the types of the involved variables. For this reason, all primitive functions applicable in an OOPN must be in advance specified from the point of view of the correspondence of the types of their input and output values. Next, as we work with sets of (extended) types symbolically representing multisets of tokens, the multiplicities of elements in initial markings and arc expressions are ignored. Moreover, we do not remove the (extended) types symbolically representing some tokens once they get into a place.

To ensure that the type analysis process will terminate in a finite number of steps, we further have to work around the possibility of unlimited nesting of tuples (at least when we want the type of a tuple to reflect the types of the elements of the given tuple). This problem can be solved by imposing a limit on the maximum depth of the nested tuples that we will process in an exact way. If this limit is exceeded, we can express the type of the appropriate tuple in an inexact way by means of the construct ($\texttt{\#tuple}, \mathit{set\_of\_types}$) where $\mathit{set\_of\_types}$ contains all the types of the non-tuple elements nested in the tuple. The just mentioned problem is illustrated in Fig. 2.

Finally, a similar problem to the above is associated with the length of tuples. However, we can again impose a limit on the maximum length of tuples to be precisely typed and if this is exceeded, we can use the same approximate typing construction as in the case of too deep nesting.

### 3.3 The OOPN Static Type Analysis Algorithm

In this subsection, we informally present a kernel of the OOPN static type analysis algorithm. A formal and complete description is beyond the scope of this article. More details can be found in [7, 4].

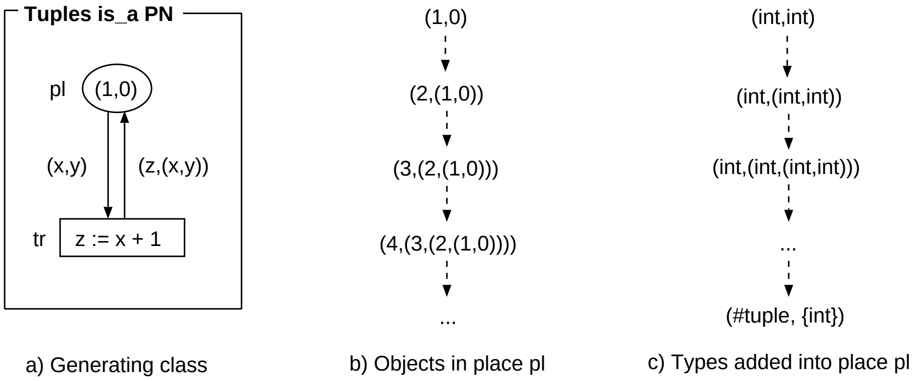



**Fig. 2:** A solution of the problem of the types of nested tuples

We have already mentioned that the static type analysis algorithm deals with *symbolic markings* which mark particular places by sets of types instead of multisets of concrete tokens. The structuring of states into net instances is ignored.

Let us further recall that the dynamic behaviour of OOPNs is based on the notion of events. An event describes firing a transition in some net instance under some binding of the variables that appear on the arcs and in the guard expressions of this transition and of the adjoined ports. In the following, we work with the so-called *symbolic events* and *symbolic bindings* which differ from events and bindings in such a way that all the involved constants, classes, and instance identifiers are replaced by the corresponding types. A symbolic binding yields a symbolic `N` event over a transition if the keyword `new` occurs in the action of the transition and the appropriate symbolic receiver is the metaclass of some class $c$ (which we encode as $(\texttt{\#class}, c)$). A symbolic binding gives rise to a pair of symbolic `F` and `J` events over a transition if the appropriate symbolic receiver is an OOPN-based class which implements a method with the selector used in the action of the transition. In all other cases, a symbolic binding yields a symbolic `A` event.

If we neglect a possible lack of free identifiers, an `A`, `N`, or `F` event is enabled if there are enough tokens in the input and tested places of the appropriate transition and of the involved ports and if all the guard expressions to be trivially evaluated result in true. A symbolic `A`, `N`, or `F` event is enabled if there are some types (i.e. symbolic tokens) in all the appropriate places and these types are compatible with the appropriate arc expressions (`int` is e.g. not compatible with `#e` or `(x,y)`) and with the types of the involved trivial functions. In the case of a symbolic `A` event, the variable containing the result of the appropriate transition action has to be bind to the type corresponding to the prototype of the involved trivial function (for example, $\texttt{int+int} \Rightarrow \texttt{int}$). In the case of a symbolic `N` event, the result variable should contain the class being instantiated.

A regular `J` event is enabled if there is a token in the return place of the method net instance invoked from the appropriate transition. A symbolic `J` event is enabled if the appropriate transition is symbolically `F` enabled and there is some type in the appropriate return place. This type has to be assigned to the variable which should contain the result of the appropriate computation.

To accelerate the computation, we will further work with sets of the so-called *ascendent transitions* of particular places. We will denote these sets by $AT_p$ and use them to find out the transitions whose symbolic firability should be re-examined after a change in the symbolic marking of $p$ (instead of re-examining all transitions). The set $AT_p$ of a place $p$

will include all the transitions that are linked with $p$ by an input or testing arc and also all the transitions that may (possibly transitively) call some synchronous port linked to $p$ by an input or testing arc. In the case of return places, $AT_p$ has to also include all the transitions that can call the appropriate method. The ability of calling a port or a method is to be decided statically according to the use of its selector.

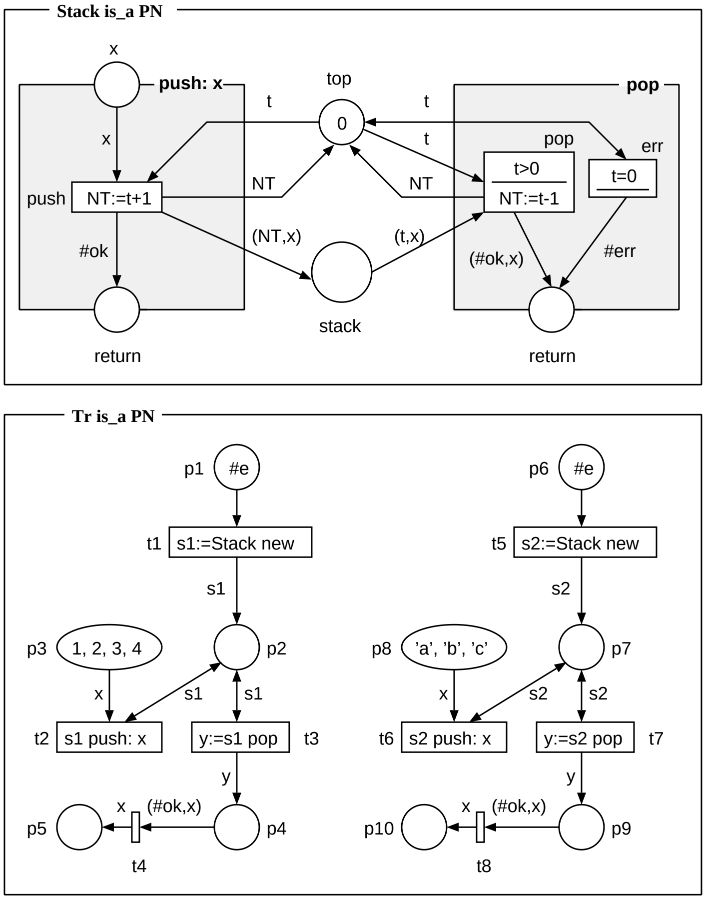
The procedure of the static type analysis over some OOPN can now be described as the following sequence of steps:

1. (a) Unfold the inheritance hierarchy not to mix types of places inherited to different classes.
   (b) Remove multiplicities from arc expressions and initial markings. We will work with sets instead of multisets from now on.
   (c) Replace all constants in arc expressions and initial markings by their types. All classes replace by the special type $(\texttt{\#class}, ClassName)$. Below, we will work with types instead of concrete values. Inscription variables are not modified.
   (d) For each place $p$, let $TS_p$ be the set of the types of tokens that may appear in $p$ in any of the corresponding net instances. Initially, each $TS_p$ is equal to the set of the types obtained from its modified initial marking.
   (e) Let $CTS$ be a set of "critical" transitions. Initially, $CTS$ contains all the transitions symbolically enabled in the initial symbolic marking.
   (f) Compute $AT_p$ for each place $p$.
2. If $CTS = \emptyset$, terminate the computation—the set $TS_p$ of each place $p$ contains the types of the tokens that are not excluded to appear in $p$. Otherwise, go to the step 3.
3. Select a transition $t$ from $CTS$, let $CTS := CTS \setminus \{t\}$, and process $t$ as follows:
   (a) Compute the set $EV_t$ of the symbolically enabled symbolic events based on $t$.
   (b) Process each $E \in EV_t$ according to its type:
      i. `A`, `N`: Propagate the types from the involved symbolic binding along the output arcs of $t$ and of the adjoined synchronous ports.
      ii. `F`: Add the types of the arguments obtained from the involved symbolic binding $b$ into $TS_p$ of the input places of the invoked method. Propagate the types from $b$ along the output arcs of the synchronous ports activated from $t$.
      iii. `J`: Propagate the types from the involved binding along the output arcs of $t$.
   (c) For each place $p$ whose $TS_p$ was changed in the previous step, add all the transition from $AT_p$ into $CTS$.
4. Go back to the step 2.

### 3.4 From the Static to the Dynamic OOPN Type Analysis

The static type analysis algorithm is relatively fast, but the obtained results may be quite inexact. Apart from dealing with sets of types instead of multisets of concrete tokens and apart from not removing types from places once they get there, there is the problem that we entirely ignore the structuring of running models into individual net instances. Consequently, various types may be propagated to places where they could get only if it was possible to store a token into one instance and read it from another instance.

The above problem is illustrated in Fig. 3 that presents a simple model of a stack. The problem manifests itself such that both `p5` and `p10` are typed by $\{\texttt{int}, \texttt{char}\}$ although only integer-coloured tokens can get to `p5` and only character-coloured ones to `p10`.




a) An OOPN-based model of a stack

| Place | Set of types |
|---|---|
| top | {int} |
| stack | {(int,int), (int,char)} |
| push: x | {int, char} |
| push: return | {symbol} |
| pop: return | {symbol, (symbol,int), (symbol,char)} |
| p1 | {symbol} |
| p2 | {Stack} |
| p3 | {int} |
| p4 | {symbol, (symbol,int), (symbol,char)} |
| p5 | {int, char} |
| p6 | {symbol} |
| p7 | {Stack} |
| p8 | {char} |
| p9 | {symbol, (symbol,int), (symbol,char)} |
| p10 | {int, char} |

b) The results of the static type analysis of the model

**Fig. 3:** An illustration of the inexactness of the OOPN static type analysis

It is obvious that the just mentioned disadvantage of the OOPN static type analysis is especially related to dealing with data structures, such as stacks, queues, lists, tables, arrays, etc. However, it can also be encountered in OOPNs that do not use structures of this kind at all. If this feature of the OOPN static type analysis is not acceptable, we have to apply some other kind of type analysis, may be the further mentioned dynamic OOPN type analysis.

The OOPN dynamic type analysis [7] attempts to distinguish particular instances to some degree according to the history of their creation. We use three pieces of information to identify a given symbolic net instance and its tokens: *oid*, *pid*, and *bid*. The *oid* represents the identifier of the object to which the given net instance belongs. The value of *pid* specifies the identifier of the net instance itself. In the case of an object net instance, $oid = pid$. However, both *oid* and *pid* are inexact—they are represented by sets containing unique integer identifiers of all the transitions that participated in the creation of the appropriate net instance in a chain of instance creations beginning in the initial object. For the initial object itself, we have $oid = pid = \{0\}$. As for the *bid*, this is used for distinguishing method net instances that are started over the same symbolic object from the same symbolic transition instance, but with different symbolic bindings. In the case of object nets, *bid* always equals to 1. Of course, OOPN inscriptions have to be suitable modified to work with the described identification of instances—for details see [7].

So, it is obvious that the OOPN dynamic type analysis does really not identify instances in an exact way. The method neglects possible cycles in the processes of instance creation. Moreover, it also does not take into account the order of creating instances. Nevertheless, the chosen approach seems to be a promising compromise between the computational complexity and the quality of the obtained results. For example, the OOPN dynamic type analysis method can successfully distinguish the two instances of the class `Stack` in Fig. 3. Consequently, the dynamic method can derive the types of places in the stack example in an exact way.

## 4 Conclusion

In the paper, we have discussed the problem of automatically deriving types in the context of the weakly typed OOPNs associated with the language PNtalk. The type analysis can be used to debug models, to improve their documentation, and to optimise the representation of OOPNs to be used for running and/or analysing and verifying models based on OOPNs. We have briefly presented two methods for an automatic OOPN type analysis, namely the so-called static and dynamic OOPN type analysis. In the near future, the described type analysis methods will be implemented over the new Prolog-based kernel of the PNtalk system, which will allow us to obtain some practical experience from their application.

*This work was done within the project CEZ:J22/98: 262200012 "Research in Information and Control Systems" and it was also supported by the Grant Agency of the Czech Republic under the contract 102/00/1017 "Modelling, Verifying, and Prototyping Distributed Applications Using Petri Nets".*

## Bibliography

1. M. Češka, V. Janoušek, and T. Vojnar. PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling. In F. Pichler and R. Moreno-Díaz, editors, *Computer Aided Systems Theory and Technology – EUROCAST'97*, volume 1333 of *Lecture Notes in Computer Science*, pages 591–610, Las Palmas de Gran Canaria, Spain, February 1997. Springer-Verlag.
2. M. Češka, V. Janoušek, and T. Vojnar. Generating and Exploiting State Spaces of Object-Oriented Petri Nets. In M. Pezzé and S.M. Shatz, editors, *Proceedings of Workshop on Software Engineering and Petri Nets*, DAIMI PB-548, pages 35–54, Aarhus, Denmark, 2000. Department of Computer Science, University of Aarhus.
3. V. Janoušek. *Modelling Objects by Petri Nets.* PhD thesis, Department of Computer Science and Engineering, FEECS, Technical University of Brno, Czech Republic, 1998.
4. B. Křena. Type Analysis in Object-Oriented Petri Nets. Technical report, Dept. of Computer Science and Engineering, FEECS, Technical University of Brno, Czech Republic, 2001.
5. The Petri Nets World, 2001. An on-line database of Petri net-related conferences, mailing lists, bibliographies, tool databases, newsletters, research groups, etc. URL: `http://www.daimi.au.dk/PetriNets/`.
6. A. Valmari. The State Explosion Problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
7. T. Vojnar. Evaluating Time-Dependent Properties of Models Described by Object-Oriented Petri Nets. Ph.D. thesis proposal, Department of Computer Science and Engineering, Technical University of Brno, Czech Republic, 1998.