MODELLING, PROTOTYPING, AND VERIFYING CONCURRENT AND DISTRIBUTED APPLICATIONS USING OBJECT-ORIENTED PETRI NETS

Milan Češka, Vladimír Janoušek, Tomáš Vojnar
Department of Computer Science and Engineering, Brno University of Technology
Božetěchova 2, CZ-612 66 Brno
e-mail: {ceska, janousek, vojnar}@dcse.fee.vutbr.cz

object-orientation, Petri nets, distributed applications, rapid prototyping, formal methods

ABSTRACT

This paper presents several research issues associated with the PNtalk language that is based on a certain kind of object-oriented Petri nets (OOPNs) and intended mainly for modelling, prototyping, and verifying concurrent and distributed applications. The paper reviews the main concepts of PNtalk and OOPNs followed by a proposal of a system allowing prototypes based on PNtalk to be run in a distributed way. Furthermore, the first steps made towards state spaces-based formal analysis and verification over PNtalk OOPNs are also briefly mentioned in the paper.

1. INTRODUCTION

This paper reviews some of the research activities related to the object-oriented Petri nets (OOPNs) associated with the language and tool called PNtalk (Češka and Janoušek 1997; Janoušek 1998), which have been developed mainly to support modelling, investigating, and prototyping concurrent and distributed object-oriented software systems. PNtalk supports intuitive modelling of all the key features of these systems, including object-orientedness, message sending, parallelism, and synchronisation. The main modelling features of PNtalk and OOPNs are presented at the beginning of this paper.

The use of PNtalk should be supported by a tool suite constituting the so-called PNtalk system. A prototype of such a system supporting modelling and simulation of systems by means of OOPNs was already implemented some time ago (Češka *et al*. 1997). Here, we sketch the main principles of a new architecture of the PNtalk system that should allow us to run OOPN-based prototypes in a truly distributed way and with the possibility of creating mobile PNtalk objects.

In the remaining part of this paper there are briefly presented some issues related to generating and using state spaces of OOPNs for formal analysis and verification of the considered systems. These issues are mostly related to the fact that we have to deal with dynamically arising and disappearing instances in state spaces of OOPNs (Vojnar 2001).

2. PNtalk AND THE ASSOCIATED OOPNs

The OOPN formalism associated with PNtalk is characterized by a Smalltalk-based object-orientation enriched with concurrency and polymorphic transition execution, which allows for message sending, waiting for and accepting responses, creating new objects, and performing primitive computations (Janoušek 1998).

An example illustrating the notation of PNtalk is shown in Fig. . It is a fragment of a PNtalk class whose each object represents a session of a user working with a certain graphic object in the system of a cooperative editor for hierarchical diagrams (Bastide *et al*. 1996; Kočí and Vojnar 2001). We will return to this model in slightly more detail later.

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 (Češka *et al*. 1997) and the formal definition of OOPNs in (Vojnar 2001).

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

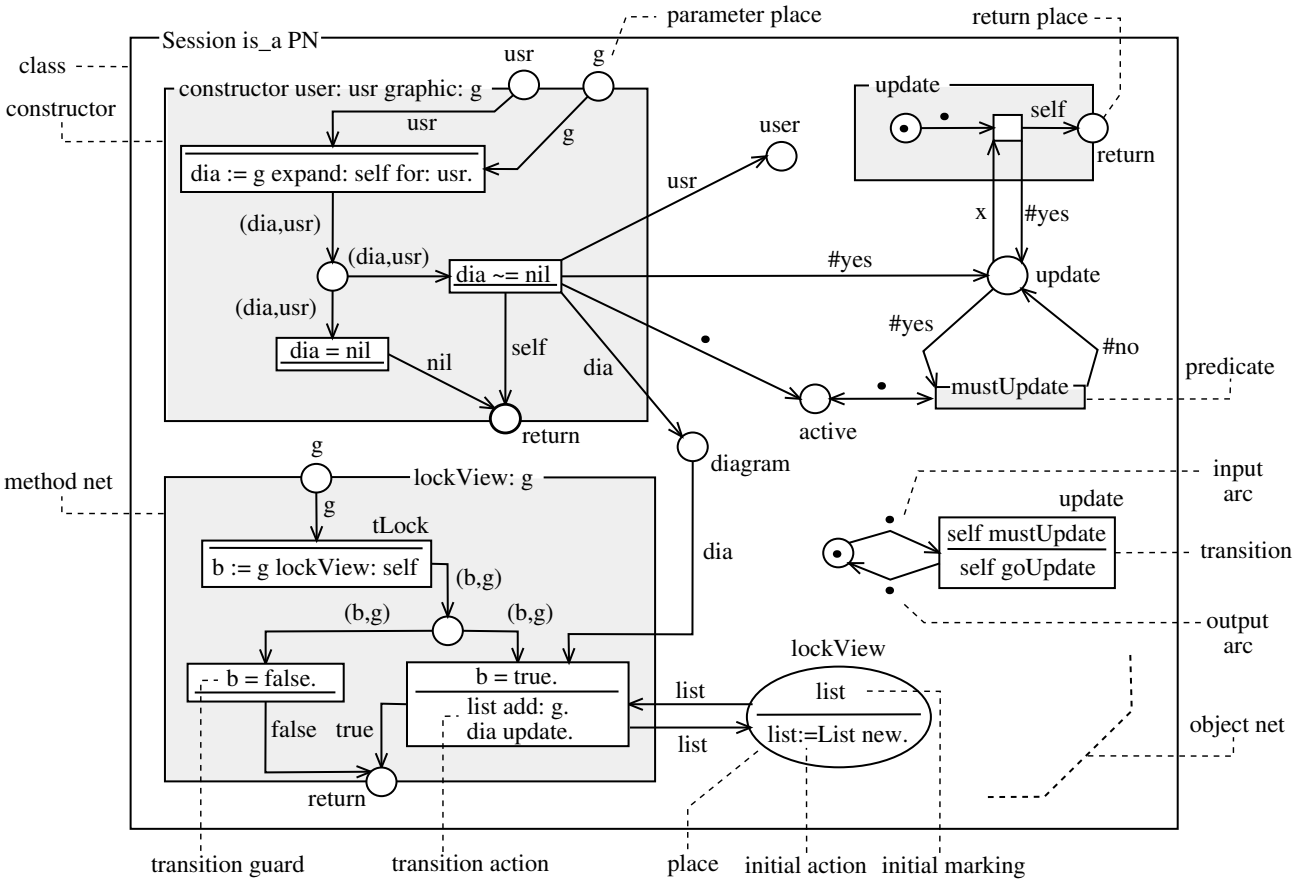

Figure 1: A fragment of a class demonstrating the notion of PNtalk

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. *Constructors* are method nets intended for initializing objects.

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. (An invocation of a constructor over a class leads to a sequence built of an `N`, `F`, and `J` event.)

2.3 An Example of an OOPN

In Fig. there is presented a fragment of the class `Session` from the system of a cooperative editor for hierarchical diagrams (Bastide *et al*. 1996; Kočí and Vojnar 2001). Let us now return to this class for a moment.

In the model of the locking mechanisms of a cooperative editor from (Kočí and Vojnar 2001), if a user wants to view or edit a diagram, the user first has to create a `Session` over the diagram. The session may be created by means of the constructor `user:graphic:` whose second parameter has to be the graphic that is the root of the diagram to be opened (in the hierarchy of diagrams managed by the editor). The graphic must be owned for encapsulation by the user. The ownership for encapsulation is checked via the method `expand:for:` invoked over the graphic. This method returns either a reference to the appropriate diagram or `nil`. The latter case causes the `Session`'s constructor to also return `nil` leading to a subsequent deletion of the not successfully created `Session` by the garbage collector.

When a diagram is changed, it sends a message `update` to all the `Session`s currently opened over this diagram. The fact that `update` is a method allows the appropriate message to be processed in an asynchronous way and makes it easier to implement this mechanism in a distributed environment. The method `update` ensures that there eventually appears a token `#yes` in the place `update` which enables the transition `update` via the synchronous port `mustUpdate`. The transition `update` calls the method `goUpdate` that implements the internal details of updating a diagram—the method is not presented here. Note that more requests to update may lead to a single physical update of the appropriate diagram.

3. TOWARDS A DISTRIBUTED PNtalk

The current version of the PNtalk system that is available on the Internet since 1996 can hardly be considered anything more than a demonstration of the concept of OOPNs. It is based on the 1995 version of OOPNs (Janoušek 1995), and so it does not reflect some of the new features of OOPNs as introduced in (Janoušek 1998), which is especially the case of synchronous ports. Moreover, the closedness and poor interoperability of this PNtalk implementation does not allow for experimenting with applications that are more than some "school examples" (such as different variants of the system of dining philosophers or very simple workflow systems).

Currently we are developing a new PNtalk system architecture that is intended for experiments with prototyping more realistic applications. This new PNtalk has to support a seamless evolution of an application from a simple model to a prototype running in real time and, possibly, to final implementation including some parts realized in PNtalk. This goal may be achieved by using encapsulation of model components allowing them to be developed independently of each other. The employed encapsulation and object-orientation have to be platform-independent as well as programming language- and network distribution-independent. Although we prefer implementations in the Smalltalk and Prolog languages, the language independence of components, as well as the network transparency, can be achieved by using some standards for inter-object communication as, e.g., CORBA. Then, each CORBA-compliant component could be connected to an OOPN-based prototype and vice versa.

The new PNtalk is not being designed in a form of a compact, integrated environment supporting editing, compiling, simulation, and debugging. It is a set of simple tools, instead. These tools can be used either independently or combined in order to pragmatically support all phases of a model or prototype development. In the early phases, using some integrated environment could be advantageous. In this case, such an environment could be built upon the mentioned tools in a form of a special software layer, as it will be mentioned later.

As the new PNtalk system implementation is not fished yet, it has to be stressed that the PNtalk system itself has to be evolvable. Its architecture proposal takes this into account—it is based on the idea that complex systems evolve from simpler ones. The simplest form of the new PNtalk system is based on a number of universal components which themselves could evolve:

An OOPN interpreter. This is a key component. It is intended to be used in all phases of a model or prototype development. It could also participate in the final implementation of a developed software system. The key parts of the interpreter may also be used for state space generation in verification tools. The interpreter should have a form of a library available in several programming languages. The very first versions are be-

ing implemented in Prolog and Smalltalk. Moreover, it should be also available as a program which it is possible to communicate with via TCP/IP. This means that the interpreter can become a part of an arbitrary software system, even a distributed one. The OOPN interpreter can be viewed as a virtual machine whose behavior is defined by the image that it works with. The image is a set of all the currently living objects in a binary form. It contains both compiled class definitions and methods as well as states of all the objects in the form of tokens distributed in Petri nets.

A compiler compiles the textual form of the PNtalk language into a binary form acceptable by the interpreter. It may work incrementally. In such a case, it embeds (possibly in cooperation with some supporting tools) the compiled code into an existing image. The compiler could be implemented as a library for some programming languages as well as a standalone program.

A source code repository. In its simplest form, it could be a file containing all the source code of the classes of the objects contained in an image. More interestingly, it could be a log file containing all changes to the sources of an image, which resembles Smalltalk. In its even more sophisticated form, it could be implemented as a database accessible via TCP/IP. The compiler has to be able to compile the source code of a method independently of the employed realisation of the source code repository.

An editor should allow us to create and modify OOPN classes and methods both in a textual and a graphical way. It interacts with the source code repository. It could possibly interact with the compiler in order to check the source code. The graphical version of the editor has to be able to load the source code obtained from the text editor and to semi-automatically add the missing graphical information to the OOPN diagrams. The source code repository should contain the textual form of OOPN components definitions together with the graphical information associated with the appropriate diagrams if it is available.

A shell (debugger) allows a programmer to interact directly with the OOPN interpreter via a TCP/IP connection. It makes it possible to load an image or to add some compiled code or saved object state to the active image. The shell makes it also possible to save the current image or its part to a file or to send it through a network connection to a file or another interpreter. Moreover, it allows a programmer to send a message to an object living in the active image and to inspect the result. It also allows him or her to list all processes, to stop execution of some process and to trace it or let it proceed. The shell has to have both a command-line as well as a graphical form. In both cases, it has to allow for tracing processes and inspecting states of objects on the source code level (textual or graphical). This means that it should also interact with the source code repository.

Analysis and verification tools should be available in the form of libraries for some programming languages as well as in the form of a server accessible via TCP/IP. The analysis and verification suite should comprise its own shell making it possible to interactively specify what is to be analysed or verified.

All these tools constitute the basic level of the PNtalk system components. These components can interoperate with other software systems. They also allow for a simple form of communication over TCP/IP networks. On top of this basic level components, a distributed PNtalk system could be built in several steps:

A simple client-server PNtalk architecture. The client integrates the editor and shell (debugger) enriched with the possibility of invoking the compiler remotely. It connects via HTTP to a server that encapsulates reentrant interpreter, compiler, and analysis and verification tools. All functionality of these tools is offered to each authenticated user. This architecture can exploit powerful computational servers on the network. It allows multiple programmers to develop independent prototypes that can communicate via TCP/IP.

A multiuser PNtalk. Several authenticated users or client applications may interact with a shared PNtalk interpreter. Each user owns some objects and these objects have access rights defined. We obtain a client-server PNtalk that enables a tighter coupling of components developed by various programmers. The shared PNtalk interpreter is a modified version of the basic PNtalk interpreter described above.

A distributed PNtalk. The PNtalk interpreter is modified in such a way that objects can communicate with each other regardless the interpreter they are living in. This means that the image can be distributed among several interpreters and the distribution is transparent to the objects. To support this mode of PNtalk operation, some name server has to be introduced, allowing an object identified by some oid to be located. The interpreter has to decide where the receiver of the processed message lives and to possibly delegate the message delivery to some other interpreter. The distributed PNtalk can also be accessed in a client-server manner in which case the

server is built upon a cluster of PNtalk interpreters, a name server, a source code repository, a compiler, and a verification server.

Note that the Multiuser PNtalk and the Distributed PNtalk are mutually independent modifications of the basic PNtalk interpreter. Nevertheless, they can be merged.

4. ANALYSIS AND VERIFICATION OVER OOPNs

In this section, we briefly mention the first steps made towards exploiting formal analysis and verification methods in the context of OOPNs (Vojnar 2001). Using formal analysis and verification can be considered complementary to validating systems by simulation because although we need not be able to fully verify or analyse the behaviour of a system, even partial analysis or verification can reveal errors different from the ones found by simulation. Among the different methods of performing formal analysis or verification, generating and exploring suitably represented state spaces–see, e.g., (Valmari 1998)—appears to be the most straightforward approach for the case of OOPNs.

4.4 Generating State Spaces of OOPNs

In state spaces of OOPNs, it appears necessary to be beware of the so-called *naming problem* (Vojnar 2001). This phenomenon corresponds to the undesirable possibility of generating many states differing only in the identifiers of the involved net instances. Two causes of the naming problem in the domain of state spaces of OOPNs (and other formalisms with dynamic instantiation) may be identified: (1) assigning names to newly arising instances unnecessarily reflecting the history of their creation and (2) dealing with concurrently existing, uniquely identified instances that play somehow symmetrical roles in the appropriate models (as a reflection of some symmetries existing already on the level of the modelled systems). Although the naming problem is not exclusively specific to OOPNs (nor to formalisms with dynamic structuring), it manifests itself in an especially severe way here due to the first mentioned source specific to this area.

In (Vojnar 2001) there have been proposed and compared two methods for dealing with the naming problem in the context of OOPNs, namely *sophisticated naming rules* and *name abstraction*. The idea of using sophisticated naming rules is inspired by the approach to identifying processes in Spin (Holzmann 1997), which has been generalized and modified to suit better the needs of object-orientation and Petri nets. Name-abstraction, on the other hand, is a fully transparent application of the concept of symmetrically reduced state spaces—see, e.g., (Ip and Dill 1996; Junttila 1999)—to solving the naming problem. When applying symmetries to solving the naming problem in the context of state spaces of OOPNs, it is necessary to take into account some special issues, such as: garbage collection, encapsulation of method net instances in objects, infinite numbers of the possibilities how to identify a newly arising instance, or providing a sufficiently broad notion of trivial operations over instance identifiers. The most elaborated sophisticated naming rules appear to be applicable when dealing with systems without many system-level symmetries, especially when partial order reduction is used. In the other cases (which do not seem to be only exceptional when applying object orientation), using name abstraction can be more advantageous.

Let us add that the naming problem should always be solved together with the problem of *removing unnecessary instances from states*. This is because that if we do not systematically remove such instances, the number of concurrently existing instances may grow and the naming problem can manifest itself stronger. Moreover, there can be generated redundant, semantically equal states distinguished by the different garbage present in them even when we apply a very good solution of the naming problem. Therefore it seems to be advantageous to remove all unnecessary instances from states as soon as possible. In the case of the PNtalk OOPNs, we use an *immediate garbage collecting mechanism* to solve the above problem. In order for this mechanism to work properly, it suffices to create OOPNs such that they do not store references to obsolete instances.

4.5 Using State Spaces of OOPNs

In the area of formal analysis and verification of concurrent systems there have already been proposed many ways of expressing properties of the systems under investigation to be evaluated over state spaces of their models (Valmari 1998). They include, e.g., using state space statistics, universal state space query languages, property labels or directives, or temporal logics. Most of the common ways of *asking state space analysis or verification questions* can be accommodated for dealing with OOPN-based models too (Vojnar 2001). Properties to be evaluated over state spaces of OOPNs should not refer to the concrete identifiers of instances. This is because the concrete names of instances do not normally have any influence upon the behaviour of OOPN-based models, and, moreover, it is hard (and sometimes impossible) to predict what identifiers will be used for what instances in what states.

Some new useful kinds of system properties to be checked may also be introduced in the area of systems that can be suitably described by OOPNs—e.g., persistence of instances, instance-oriented progress, etc. Generally, analysis or verification questions to be answered over state spaces of OOPNs

can ignore or, on the other hand, respect the structuring of running OOPNs into instances. Unfortunately, dealing with instance-oriented analysis or verification queries may lead to higher time and space requirements (Vojnar 2001). This applies especially in the cases when it is not sufficient to distinguish particular instances within a single state, and we have to track their individual behaviour along some state space paths.

The particular specification and query techniques can be applied in the domain of OOPN-based models such that analytical or verification questions described by them could be answered over (possibly name-abstracted) state spaces of OOPNs by means of fairly standard algorithms. In some cases, however, the standard algorithms have to be slightly modified to a certain degree to be able to deal with models structured into sets of net instances and objects, which can be dynamically created and/or discarded (Vojnar 2001).

5. CONCLUSIONS

In the paper, we have presented several research issues related to the PNtalk language and system and to the OOPNs associated with them.

Firstly, we have outlined the main concepts of the new architecture of the PNtalk system which should allow us to run OOPN-based prototypes in a truly distributed way. Although this architecture is still subject to development, some parts of the new PNtalk system have already been experimentally implemented (a new PNtalk interpreter, a simple client-server variant of the PNtalk system). Now, the distributed architecture of PNtalk is to be further refined, fully implemented, and tested on a suitable case study—for this reason, we have decide to use the cooperative editor case study (Bastide *et al*. 1996; Kočí and Vojnar 2001).

We have also presented the first steps which have been done in the area of generating and using state spaces of OOPNs for formal analysis and verification. A prototype of a generator of state spaces of OOPNs (using name abstraction or, alternatively, sophisticated naming rules) and a processor of a simple state space query language have already been built. In the future, we intend to improve the current state space tool, to augment it with an option of partial-order reduction, and to try to enable export to some already existing state space tools (possibly slightly extended). We have proposed a technique for automatic type analysis in OOPNs (Křena and Vojnar 2001) as well.

Acknowledgement. This work was done within the project CEZ:J22/98:262200012 “Research in Information and Control Systems” and within the project of the Grant Agency of the Czech Republic No. 102/00/1017 “Modelling, Verifying, and Prototyping Distributed Applications Using Petri Nets”.

REFERENCES

Janoušek, V. 1995. “PNtalk: Object Orientation in Petri nets”. In: *Proceedings of European Simulation Multiconference ESM’95*. Czech Technical University, Prague, pp. 196–200.

Bastide, R.; C.A. Lakos; and P. Palanque. 1996. *A Cooperative Petri Net Editor*. A case study proposal for the 2nd Workshop on Object-Oriented Programming and Models of Concurrency, ICATPN’96, Osaka. URL: `http://wrcm.dsi.unimi.it/PetriLab/ws96`.

Češka, M. and V. Janoušek. 1997. “A Formal Model for Object-Oriented Petri Nets Modeling”. *Advances in Systems Science and Applications, An Official Journal of the International Institute for General Systems Studies*, Special Issue, pp. 119–124.

Češka, M.; V. Janoušek; and T. Vojnar. 1997. “PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling”. In: *Computer Aided Systems Theory – EUROCAST’97* (Las Palmas de Gran Canaria, Spain). Springer-Verlag, LNCS, vol. 1333, pp. 591–610.

Holzmann, G. 1997. “The Model Checker Spin”. *IEEE Transactions on Software Engineering*, vol. 23(5).

Ip, C. and D. Dill. 1996. “Better Verification Through Symmetry”. *Journal of Formal Methods in System Design*, vol. 9(1/2), pp. 41–76.

Janoušek, V. 1998. “Modelling Objects by Petri Nets”. PhD thesis. Dept. of Computer Science and Eng., FEECS, Brno University of Technology, Czech Republic. (In Czech).

Junttila, T. 1999. “Detecting and Exploiting Data Type Symmetries of Algebraic System Nets during Reachability Analysis”. Technical Report HUT-TCS-A57. Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland.

Kočí, R. and T. Vojnar. 2001. “A PNtalk-based Model of a Cooperative Editor”. In: *Proceedings of MOSIS’01* (Hradec nad Moravicí, Czech Republic). MARQ Ostrava, pp. 165–172.

Křena, B. and T. Vojnar. 2001. “Type Analysis in Object-Oriented Petri Nets”. In: *Proceedings of ISM’01* (Hradec nad Moravicí, Czech Republic). MARQ Ostrava, pp. 173–180.

Valmari, A. 1998. “The State Explosion Problem”. In: *Lectures on Petri Nets I: Basic Models*, edited by W. Reisig, G. Rozenberg. Springer-Verlag, LNCS, vol. 1491, pp. 429–528.

Vojnar, T. 2001. “Towards Formal Analysis and Verification over State Spaces of Object-Oriented Petri Nets”. PhD thesis. Dept. of Computer Science and Eng., FEECS, Brno University of Technology, Czech Republic.