Generating and Using State Spaces of Object-Oriented Petri Nets *

Milan Češka, Vladimír Janoušek, and Tomáš Vojnar

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

Abstract. The article describes two of the problems that arise in the area of generating and exploiting state spaces of the object-oriented Petri nets (OOPNs) which underlie the modelling language used in the tool called PNtalk. We first explain the influence of identifiers of dynamically appearing and disappearing instances upon the state space explosion problem. We describe and compare methods of working with identifiers based on sophisticated naming rules and mechanisms for abstracting names. Subsequently, there are discussed several approaches to specifying properties of systems to be validated by means of state spaces of their OOPN-based models.

1 Introduction

Current complex distributed applications require dealing with dynamically arising and disappearing objects which can communicate, synchronize their actions, and migrate among particular nodes of the distributed environment they are running in. This kind of applications include distributed operating systems, groupware allowing a concurrent work of several people on the same project, applications exploiting the technology of (mobile) agents, etc.

A language called PNtalk based on object-oriented Petri nets (OOPNs) has been developed at the DCSE, TU Brno in order to support modelling, investigating, and prototyping complex distributed object-oriented software systems [8]. PNtalk supports intuitive modelling all the key features of these systems, such as object-orientedness, message sending, parallelism, and synchronization. This is achieved through working with active objects encapsulating sets of processes described by Petri nets. Processes inside the objects communicate via a shared memory, while objects themselves communicate by message passing.

Simulation is one way of examining systems modelled by OOPNs and it is already supported by a prototype version of a tool called PNtalk [2]. Moreover, models of software systems created in PNtalk should be usable as prototypes of these systems, as well. Currently we are working on new implementations of PNtalk in Prolog and in Smalltalk which should allow us to run OOPN-based prototypes in a truly distributed way.

Although we have started with simulation and prototyping, this article concentrates on the first steps made towards exploiting formal analysis and verification methods in the context of OOPNs. This approach can be considered complementary to simulation because although we are not always able to fully verify or analyse the behaviour of a system, even partial analysis or verification can reveal some errors which tend to be different from the ones found by simulation [15]. We believe that object-orientation should allow us to relatively easily extract the subsystems to be verified and to abstract their environment.

Among the different approaches to performing formal analysis or verification, generating and exploring suitably represented state spaces appears to be the most straightforward way for the case of OOPNs. Methods based on state spaces are quite universal, can be almost fully automated, and allow a relatively easy implementation. There have been proposed

* This paper is an extended and slightly modified version of [3].

many different ways of alleviating their main deficiency—the state explosion problem [15]. We believe that many of these methods can successfully be adapted and optimized for dealing with state spaces of highly dynamically structured OOPNs, as well.

In this article, we show that when we want to work with state spaces of OOPNs (or more generally of any formalism with dynamic instantiation of some kind of components), we have to pay a careful attention to treating identifiers of objects (or in general some other kind of dynamically appearing and disappearing instances). Otherwise, many unnecessary states can be generated and the state spaces can even unnecessarily grow to infinity. This naming problem can be solved either by introducing some sophisticated rules for assigning identifiers to instances or by not considering concrete names of instances to be important when testing states to be equal (which is an application of the symmetry reduction method [9, 7, 13]).

The object-oriented structuring of OOPN-based models influences not only the generation of the appropriate state spaces, but also the use of such state spaces. This is because it appears to be useful to have tools for investigating not only properties related to global states of OOPNs, but also to individual dynamically arising and disappearing instances. At the same time we should be able to ask analysis and verification questions without referring to the concrete names of particular instances. These names are semantically not important and, what is more, modellers can hardly work out what identifiers will be used in different states.

In the following, we first present the main ideas behind the OOPN formalism. Subsequently we discuss possible solutions of the naming problem arising in state spaces of OOPNs. Finally, we discuss several possible approaches to asking analysis or verification questions to be answered over states spaces of OOPNs.

2 Key Concepts of OOPNs

The OOPN formalism is characterized 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 [2, 8]. An example illustrating the notation of OOPNs is shown in Figure 1.

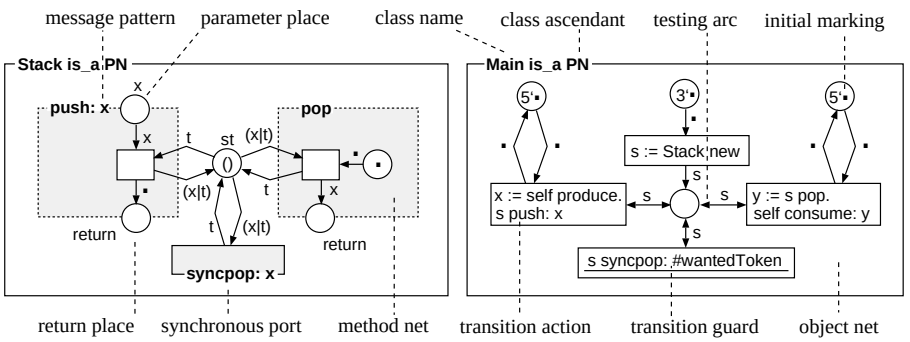

Fig. 1. An OOPN example (`Main`'s methods `produce` and `consume` are not shown).

The main principles of the structure and behaviour of OOPNs are explained in the following. However, due to space limitations the description presented is not formal or complete. We explain the necessary notions, only. A bit deeper introduction to the OOPN formalism can be found in [2] and the complete definition of OOPNs in [8, 16].

2.1 The Structure of OOPNs

An *object-oriented Petri net* is defined on a collection of elements comprising sets of constants $CONST$, variables VAR, net elements (i.e. places PL and transitions TR), class elements (i.e. object nets $ONET$, method nets $MNET$, synchronous ports $SYNC$, and message selectors MSG), classes $CLASS$, object identifiers OID, and method net instance identifiers MID. (We denote $NET = ONET \cup MNET$ and $ID = OID \cup MID$.) An OOPN has its initial class c_0 and the name oid_0 of its initial object from c_0, as well. The so-called universe U of an OOPN contains (nested) tuples of constants, classes, and object identifiers. The set of all possible bindings of variables of an OOPN is denoted as $BIND$, $BIND = \{b \mid b : VAR \longrightarrow U\}$.

Object nets of OOPNs consist of places and transitions. Each place has some—possibly empty—initial marking. Every transition has conditions (i.e. inscribed testing arcs), preconditions (i.e. inscribed input arcs), a guard, an action, and postconditions (i.e. inscribed output arcs). *Method nets* are similar to object nets but, in addition, each of them has a set of parameter places and a return place. Method nets can access places of the appropriate object nets in order to allow 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.

An OOPN *class* is specified by an object net (an element of $ONET$), a set of method nets (a subset of $MNET$), a set of synchronous ports (a subset of $SYNC$), and a set of message selectors (a subset of MSG) corresponding to its method nets and ports. Object nets describe what data particular objects can encapsulate and what activity they may 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.

The inheritance mechanism of OOPNs allows an incremental specification of classes. Inherited methods and synchronous ports can be redefined and new methods and synchronous ports can be added. A similar approach applies for object net places and transitions. (A detailed description of the inheritance mechanism is beyond the scope of this paper.)

2.2 The Dynamic Behaviour of OOPNs

The dynamic behaviour of OOPNs corresponds to the evolution of a system of objects. An *object* of a certain class c is a system of net instances which 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 $id \in ID$ and a marking of its places and transitions. A *marking of a place* is a multiset of elements of the universe U. A *transition marking* is a set of invocations.[1] Every *invocation* contains an identifier $id \in MID$ of the invoked net instance and a stored binding $b \in BIND$ of the input variables of the appropriate transition.

A state of a running OOPN has the form of a *marking*. Markings of OOPNs are represented as multisets of marking elements in order to allow the classical Petri net-way of manipulating them. In the case of a transition marking, the identifier of the invoked

[1] Transition marking in OOPNs reflects the fact that OOPN transitions may invoke methods and wait for their termination. This (possibly) non-atomic execution of transitions is a special feature of OOPNs.

method net instance is stored within the appropriate binding in a special (user-invisible) variable `mid`. Thus a formal compatibility of place and transition markings is achieved and it is possible to define a marking element as a triple consisting of the identifier of the net instance it belongs to, the appropriate place or transition, and an element of the universe or a binding. Then a marking M is a multiset

$$M : (ID \times PL \times U) \cup (ID \times TR \times BIND) \longrightarrow \mathbb{N}.$$

A change of a marking of an OOPN into another marking is a result of an occurrence of some *event*. Such an OOPN event is a 4-tuple

$$E = (e, id, t, b)$$

including (1) its type e, (2) the identifier $id \in ID$ of the net instance it takes place in, (3) the transition $t \in TR$ it is statically represented by, and (4) the binding tree b containing the bindings 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 types 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. If an enabled event E occurs in a marking M and changes the marking to M', we call this a *step* and denote it by

$$M[E\rangle M'.$$

For a given OOPN, its *initial marking* M_0 corresponds to a single, initially marked object net instance from the initial class c_0 identified as oid_0. The *set of all markings* of an OOPN is denoted as MA and the *set of all events* as EV. The set of markings *reachable* from M is denoted as $[M\rangle$.

Finally, we introduce the following notation. Given $id \in ID$, $net(id)$ denotes the net $n \in NET$ such that id identifies an instance of n, and $oid(id)$ denotes $oid \in OID$ such that id identifies a net instance belonging to the object identified by oid. Note that an object is identified by the identifier of its object net instance.

3 The Role of Instance Identifiers in State Spaces of OOPNs

In this section, we discuss the so-called naming problem associated with dealing with the identifiers of dynamically appearing and disappearing instances. We introduce and compare two methods intended to minimize the impact which the presence of instance names in states has upon the state space explosion problem. From the point of view of the naming problem, OOPNs can be considered just a representative of formalisms with dynamic instantiation of some kind of components, such as objects or processes.

3.1 The Naming Problem

When dealing with OOPNs (or any other formalism with dynamic instantiation), we can use the common notion of state spaces [15] consisting of a set of states S, a set of structural transitions T, a set of semantic transitions (i.e. links between states and structural transitions) $\Delta \subseteq S \times T \times S$, and an initial state $s_0 \in S$. It is, however, necessary to pay a careful attention to the presence of instance identifiers in states.

The instances that exist in a given state of a system have to be uniquely identified to make it possible to refer to the local states of the instances and to express references among them. Working with some sort of instance identifiers, e.g. in the form of addresses of objects, is common when running object-oriented programs or simulating object-oriented

models. However, in the context of state spaces, the presence of instance identifiers in states can significantly contribute to the state space explosion problem. This is due to the possibility of unnecessarily generating many redundant states differing only in the names of the involved instances even if the names cannot influence the future behaviour of the system being examined in any way (up to renaming). Let us denote this phenomenon as the *naming problem*.

State space redundancies associated with the instance names have two possible sources. The first one is due to formalisms supporting dynamic structuring of some kind, and stems from the way these formalisms *assign identifiers to newly arising instances*. Redundant states arise when an instance playing a certain role in the modelled system (given by its type, the local states it can reach, or the way it is referred to) can be created under different names when going through different state space paths leading to its creation. To illustrate such a case, we can consider for example a model of a flexible manufacturing system with several production cells represented by objects uniquely distinguished by the machines they encapsulate. Redundant states are generated when these objects can be created and identified using different combinations of names, possibly reflecting the order in which they were independently created of each other.

Concurrent, symmetrical work with several instances is the second possible source of state space redundancies associated with dealing with instance identifiers. In this case, some symmetries that exist already on the level of the modelled system manifest themselves in such a way that the appropriate model *treats certain instances in a symmetrical way*. Thus we would not loose any information and would save some space if we could ignore the different identifiers of such instances and swap them in some points of the evolution of the modelled system. As an example, consider the symmetrical activities of the dining (distributed) philosophers in Figure 2. (In the following, we will sometimes call the just mentioned symmetries to be the *system-level* ones.)

The possibility of exploiting system-level symmetries for reducing state spaces is not specific to formalisms with dynamic instantiation. However, it is interesting that many of these symmetries manifest themselves in a similar way to redundancies caused purely by some internal mechanisms of dynamic structuring and might thus hopefully be solved together. Next, it is important to note that in state spaces of OOPNs there can arise significant naming redundancies even if there are no system-level symmetries at all. This is a result of the described difficulties of assigning names to dynamically arising instances. Sometimes, a naming mechanism can even make the state space of a semantically evidently finite-state system formally infinite—it suffices to keep creating and destroying the same instance (from a semantical point of view) identifying it using still new identifiers. Consequently, solving the naming problem seems not to be an option in the context of OOPNs (and other formalisms supporting dynamic instantiation), but a real necessity.

The naming problem can be solved either by restricting the semantics of the applied modelling language or by examining the appropriate state spaces via their suitably reduced representations. We prefer the second approach which allows us to use different methods for solving the naming problem in different contexts and to combine them with methods trying to remove other sources of state space redundancies. When reducing state spaces to get rid of what we consider to be unnecessary naming information, we have to prove that in doing this we do not lose anything or at least anything important w.r.t. the properties of the modelled systems to be analyzed or verified. In the context of OOPNs, it is important to be able to distinguish particular instances in states, to express their ownership relations, and to know how they are manipulated by events surrounding particular states. However, we are normally not interested in the concrete values of identifiers used for implementing the just mentioned mechanisms.

In this section, two methods for solving the naming problem are presented and compared: (1) using sophisticated naming rules for assigning identifiers to newly arising in-

stances inspired by the process identifier handling used in Spin [6] and (2) the so-called name abstraction as a specialization of the symmetry method for reducing state spaces [9, 13, 7]. The latter method is based on not considering concrete names of instances to be important when checking states to be equal, which leads to working with renaming equivalence classes of states rather than with the individual states. Both of these methods are discussed in the following in the context of OOPNs. However, first of all, we define full state spaces of OOPNs in order to obtain a basis for discussing reduction using one of the mentioned methods.

We can 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* [8, 16] 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.

It is interesting that although the naming problem is rather important in the domain of dynamically structured formalisms, there has not been devoted much interest to it, yet. Formal analysis and verification is supposed to run over the low-level semantics of the object-oriented Petri nets introduced in [12]. This, however, does not solve the naming problem at all. In Spin [6], the naming problem is treated by a simple set of rules for naming instances and by limiting the maximum number of instances, but an entirely satisfying solution is not obtained (as we will explain later on). In [10] there is a short note about what we call the naming problem and it is proposed to either require finite domains of identifiers (which seems not to be a very good solution) or to use "some symmetry analysis" (which is actually one of the methods that we investigate in detail here). Relatively recently there appeared a note about a symmetry-based canonical representation of the heap of the new Java PathFinder model checker [1], which seems to be similar to some of our methods.

3.2 Full State Spaces of OOPNs

Full state spaces of OOPNs can be defined using the general concept of state spaces mentioned earlier. For a given OOPN, states will correspond to reachable markings and structural transitions to applicable events. Semantic transitions will be defined in accordance to the firing rules of OOPNs. Finally, the initial state will be the initial marking, of course.

Definition 1 (Full State Spaces of OOPNs). *Let an object-oriented Petri net $OOPN$ with its set of markings MA, its initial marking M_0, and its set of events EV be given. We define the* (full) state space *of $OOPN$ to be the 4-tuple $StSp = (S, T, \Delta, M_0)$ such that:*

1. $S = [M_0\rangle$.
2. $T = \{(M_1, E, M_2) \in S \times EV \times S \mid M_1[E\rangle M_2\}$.
3. $\forall M_1, M_2 \in MA\ \forall E \in EV[(M_1, E, M_2) \in T \Leftrightarrow (M_1, (M_1, E, M_2), M_2) \in \Delta]$.

A consequence of the definition of full state spaces of OOPNs ignoring the naming problem is that when we try to create the first instance of some net whose domain of instance identifiers is infinite, we immediately obtain infinitely many possible target markings. Moreover, requiring sets of possible identifiers of nets to be finite will not solve the problem because (1) it can change the semantics of the model by artificially restricting the number of concurrently existing instances and (2) there can still be generated unnecessarily many target markings.

3.3 Using Sophisticated Naming Rules

Sophisticated rules for assigning identifiers to newly arising instances attempt to decrease the degree of nondeterminism potentially present in the management of names of dynamically arising and disappearing instances and thus to reduce the number of states we have to process. Such rules can be made a part of the semantics of the given modelling language. However, they can also be applied without being integrated into the modelling language. Then their application can be viewed as a tool for reducing state spaces and we have to show that when using them, we do not lose any important information w.r.t. what we want to analyse or verify. The proof could be based on showing that the application of such rules is in fact a "safely-redundant" implementation of the principle of (complete) name-abstracted state spaces discussed later.

The simplest nontrivial rule for naming instances is assigning identifiers according to some ordering over them. A deficiency of this technique is that when we are cyclically creating and destroying some instance, we will obtain an infinite state space[2]. A normally semantically equal finite representation of such a state space can be obtained by recycling identifiers immediately after they are released, i.e. by identifying newly arising instances by the smallest and currently not used identifiers. It is, of course, desirable to apply recycling of identifiers together with some mechanism, such as the OOPN immediate garbage collection mechanism, that will prevent obsolete instances from unnecessarily remaining in states and from causing semantically equal states look different because of the different garbage contained in them.

However, even when we use recycling of identifiers, there can still be generated many different states which are obviously semantically equal. To illustrate such a situation, let us have some configuration of instances characterized by the number of the involved instances, their types, trivial marking, and mutual relations. Further suppose that this configuration of instances can be created by several state space paths in which the particular instances are created in different orders and using various auxiliary instances with differently overlapped lifetimes. Then there can be generated several states containing the given configuration of instances and differing only in the names of some of the involved uninterchangeable instances (distinguished by their contents or by the way they are referenced). Here, the different combinations of names undesirably reflect the possible histories of creation of the appropriate instances.

The just described problem of generating unnecessary states can be partially avoided by using mutually independent sequences of identifiers for each type of instance. These sequences must, of course, be disjoint, which can be achieved for example via making types of instances a part of their instance identifiers. In such a case, if we create the first instance of a net n_1 and then the first instance of a net n_2, they will not be identified 1 and 2 (which would undesirably reflect the order in which the instances were created), but for example $(n_1, 1)$ and $(n_2, 1)$. After this optimization, the mechanism of identifying new instances can produce redundant states only within a model that enables several different orders of creating several instances of the same type, which can be less frequent.

A similar principle to the above can be used to achieve a further reduction of naming redundancies in the area of identifying method net instances. Here, we can keep assigning identifiers to method net instances independently for each transition and the binding of its variables used within some `F` event. This is possible because each method net instance is characterized by the unique transition marking element from which it is referenced to and which includes the transition and the binding used when starting the method. However, the different sequences of such method net instance identifiers have again to be disjoint.

[2] If we start creating and destroying some instance without any persistent change in the rest of the system, we will obtain infinitely many semantically equal states where the instance will be identified as e.g. n, $n+1$, $n+2$, ...

To give an idea about generating how many unnecessary states we can avoid just via adding the last optimization above to the previous ones, let us mention that in the case of the system of 3 distributed philosophers from Figure 2, we obtain 8127 states instead of 79292. However, we shall note that even when we apply all the above heuristics, there can still remain some redundancies. Generally, these redundancies stem from (1) the mechanisms of assigning names to objects from the same classes and to method net instances started under the same binding of the same transition and from (2) mapping system-level symmetries onto working with instance identifiers.

The principle exploited when identifying method net instances could be generalized to work even with objects. To achieve such an improvement of the described sophisticated naming rules, we could, for example, use some sort of static analysis of particular object and method nets to reveal that certain objects will never be referenced from net instances different from the ones in which they were created. Such unambiguously nested objects could then be identified locally within the net instances encapsulating them, and we could also try to isolate naming of such objects created by different transitions as in the case of starting methods.

The problem of generating unnecessary states, which need not be avoided even under advanced naming schemes, can be alleviated to some degree when using partial order reduction techniques [11, 15]. This is because these techniques reduce the number of paths leading to particular states and thus also the possibilities to obtain different permutations of identifiers of the involved characteristic instances. Nevertheless, the problem is not fully solved in this way as it is not always possible to choose only one interleaving out of a set of the possible ones. Moreover, sophisticated naming rules, even when combined with partial-order reduction methods, can hardly eliminate the second possible source of naming redundancies, i.e. the possibility of mapping system-level symmetries onto symmetrical actions with some instances and their identifiers.

A certain kind of sophisticated naming rules similar to the above has originally been used for solving the naming problem in the already mentioned model checker Spin [6]. Spin works with dynamically instantiable processes described in the specialized language Promela. Its processes are identified by integers and can be terminated and their identifiers recycled only in the reverse order to their creation starting with the "youngest" currently running process. This principle can be easily implemented, but, on the other hand, if there is a possibility of cyclic instantiation of processes with partially overlapped lifetimes, the method will not prevent the state space from growing to infinity. (Of course, this assumes that we do not apply artificial limits on the number of running processes.) Moreover, the fact that processes can be terminated and their identifiers recycled only in the reverse order to their allocation maybe acceptable in the context of Promela processes, but it is less suited when working with objects whose life is often independent of their creators. Finally, Spin does also not separate the mechanisms of identifying processes of different types.

In Spin there is applied a relatively simple static method of partial-order reduction in order for its application not to produce a large overhead. This method cannot guarantee avoiding all the redundancies caused by assigning names to newly arising processes. Furthermore, it will not avoid redundancies caused by working with names of instances being used in a concurrent, cyclical, and symmetrical way such that they can be exchanged in some states, but not in all of them. An example of a situation of this kind is the concurrent and symmetrical behaviour of a set of philosophers from Figure 2 in a system of distributed philosophers [8].

3.4 Abstracting Naming Information

With respect to the previous discussion, we now suggest another possible method for solving the naming problem based on not considering concrete values of names of instances to be important when checking "equality" of states. In other words, we are going to define two

markings to be equal if there exists a suitable permutation over the set of all identifiers whose application makes the states identical. As a consequence, we will replace working with particular states by working with renaming equivalence classes of them. Let us, however, note that an application of some efficient way of removing garbage from states still remains desirable here in order to avoid generating semantically equal states distinguished by the different garbage contained in them.

In the following, we will try to describe the method of name abstraction at least partially in a formal way—a fully formal description, together with proofs of the propositions, can be found in [16].

It should be noted here that the concept of name abstraction is a specialization of the general notion of symmetries [9, 7, 13] applied for reducing state explosion caused by the presence of concrete names of instances in states. Unlike general symmetries, highly specialised renaming symmetries can be used within all OOPN-based models and allow fully automated ways of treating them within generating state spaces. Name-abstracted state spaces could be described as a special case of symmetrically reduced state spaces, but we will define them directly to save some space and to get closer to their implementation. Let us add that the application of symmetries for solving the naming problem in the context of state spaces of OOPNs has to take into account some special issues, such as: garbage collection, encapsulation of method net instances in objects, or infinite numbers of possible identifiers of newly arising instances.

The idea of abstracting the naming information can only be applied due to the fact that the behaviour of OOPN-based models does not depend on concrete values of identifiers. Here, it is crucial that the definition of OOPNs does not allow using instance identifiers in expressions and that there cannot be performed trivial computations depending on concrete values of instance names. Therefore it can be proven that when starting from some state, concrete names of instances do not influence the future evolution of the appropriate OOPN in any way (up to renaming). Thus the future behaviour that a system may exhibit from some states which are equal up to renaming is not a reason for distinguishing these states.

We have said that we want two markings to be equal if there exists a suitable permutation over the set of all the identifiers allowed in the appropriate OOPN whose application makes the states identical. However, we do not accept all permutations. An acceptable permutation cannot change the identifier of the initial object, which is important for the garbage collecting mechanism. Moreover, it must preserve the information about the net and the object which an instance belongs to. Permutations that fulfil the just described requirements will be called *renaming permutations* in the following.

Definition 2 (Renaming Permutations). *Suppose we have an object-oriented Petri net $OOPN$ with its set of instance identifiers ID and its initial object identifier oid_0. We define* renaming permutations *over $OOPN$ to be the bijections $\pi : ID \leftrightarrow ID$ such that:*

1. $\pi(oid_0) = oid_0$.
2. $\forall id \in ID\ [net(id) = net(\pi(id))]$.
3. $\forall id \in ID\ [\pi(oid(id)) = oid(\pi(id))]$.

The concept of renaming permutations provides a basis for defining the so-called *renaming symmetries*, i.e. bijections on sets of markings and sets of events. Renaming symmetries preserve the structure and the trivial contents of markings and events and only change the involved identifiers. The formal definition of renaming symmetries can be obtained by a straightforward, but a little longer, extension of bijections working over identifiers to bijections over markings and events (for the exact definition, we refer the reader to [16]). We denote the renaming symmetry induced by a renaming permutation π as ϱ^π.

Now we can define two markings M_1, M_2 to be *equal up to renaming* iff there exists a renaming permutation π such that $\varrho^\pi(M_1) = M_2$. The same can be done for events.

In the following, we will denote the renaming equivalence relation by $\sim$. Members of its equivalence classes will be referred to using the "black board alphabet", i.e. $\mathbb{M}$ or $\mathbb{E}$, or via their representatives, i.e. $[M]$ or $[E]$. Finally, quotient sets w.r.t. $\sim$ will be denoted using $\sim$ as a subscript, as e.g. $MA_{\sim}$.

The notion of renaming symmetries allows us to formalize the already mentioned property that the concrete names of instances present in an OOPN state cannot influence anything else than the names of the instances that appear in the possible future behaviours starting from the given state. Such a property is crucial in the theory of symmetrically reduced state spaces.

Proposition 1. *Let us have an object-oriented Petri net $OOPN$ with its set of markings MA, its set of events EV, and the corresponding set of all renaming permutations Π. Then the following holds for every $M_1, M_2 \in MA$, $E \in EV$, and $\pi \in \Pi$: $M_1[E\rangle M_2 \Leftrightarrow \varrho^{\pi}(M_1)[\varrho^{\pi}(E)\rangle\varrho^{\pi}(M_2)$.*

Renaming symmetries allow us to propose the expected notion of *name-abstracted state spaces (NA state spaces)* of OOPNs. When generating a name-abstracted state space, concrete identifiers of instances will not be taken into account and two states or events will be considered equal if they are equal up to renaming. The definition of name-abstracted state spaces will be based again on the general concept of state spaces. However, this time states will correspond to reachable *name-abstracted markings*, i.e. equivalence classes of MA w.r.t. $\sim$, and structural transitions to useful *name-abstracted events*, i.e. equivalence classes of EV w.r.t. $\sim$.

Definition 3 (Name-Abstracted State Spaces). *Let us have an object-oriented Petri net $OOPN$ with its set of markings MA, its initial marking M_0, its set of events EV, and its renaming equivalence relation $\sim$. We define the* name-abstracted state space (NA state space) *of $OOPN$ to be the 4-tuple $NAStSp = (S, T, \Delta, [M_0])$ such that:*

1. $S = \{[M] \in MA_{\sim} \mid M \in [M_0\rangle\}$.
2. $T = \{([M_1], [E], [M_2]) \in S \times EV_{\sim} \times S \mid M_1[E\rangle M_2\}$.
3. $\forall \mathbb{M}_1, \mathbb{M}_2 \in MA_{\sim}\ \forall \mathbb{E} \in EV_{\sim}\ [(\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2) \in T \Leftrightarrow (\mathbb{M}_1, (\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2), \mathbb{M}_2) \in \Delta]$.

Let us note that NA state spaces preserve information about reachable states and events, but their interconnection is preserved only partially. NA state spaces do not contain information about which particular instances are manipulated by events when going from one state into another. This information is obviously not important when we are dealing with isolated states, but it might become necessary if we need to be able to explore sequences of states and events (even if we do not consider concrete names of instances to be important). It can for example be useful to know how a particular instance behaves within the events surrounding an NA state provided that the instance is given by its identifier within an arbitrary representative of the NA state. However, if we need the information which is lost in NA state spaces, it is not difficult to preserve it. For this reason, we define the so-called *complete name-abstracted state spaces (CNA state spaces)*.

We define CNA state spaces as labelled NA state spaces. Every NA state will be labelled by a tuple consisting of a representative marking belonging to the equivalence class represented by the NA state and a group of self-renaming permutations (i.e. permutations which map the representative marking to itself). Every structural transition will be labelled by a minimum set of tuples consisting of a representative event and a renaming permutation. We require that every representative event must be firable from the appropriate representative source marking leading to the appropriate representative target marking *after* applying the given renaming. Furthermore, every event firable from the representative source marking and leading to a marking equal up to renaming to the representative target

marking must be derivable (up to the name of an eventually newly arising instance) from some of the representative events via a permutation from the set of source self-renaming permutations. Self-renaming permutations can decrease the number of target markings we have to process [9].

In the definition of CNA state spaces, we use a predicate *eideq* which is fulfilled when applied to two "existing identifier equal" events. Such events can differ only in the identifier of the newly created object within an `N` event and in the identifier of the newly started method net instance within an `F` event.

Definition 4 (Complete Name-Abstracted State Spaces). *Suppose that we have an OOPN with its set of markings MA, its set of events EV, and its set of renaming permutations Π. We define a* complete name-abstracted state space (CNA state space) *of OOPN to be a triple $CNAStSp = (NAStSp, m, e)$ such that:*

1. *$NAStSp = (S, T, \Delta, [M_0])$ is the NA state space of OOPN.*
2. *$m : S \to MA \times 2^{\Pi}$ such that $\forall \mathbb{M} \in S\ [m(\mathbb{M}) = (M, \Phi) \Rightarrow (M \in \mathbb{M} \wedge \forall \varphi \in \Phi\ [\varrho^{\varphi}(M) = M] \wedge (\Phi, \circ)$ is a group$)]$.*[3]
3. *$e : T \to 2^{EV \times \Pi}$ such that for all $(\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2) \in T$ with $m(\mathbb{M}_1) = (M_1, \Phi_1)$ and $m(\mathbb{M}_2) = (M_2, \Phi_2)$, $e((\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2))$ is the smallest set such that $\forall E' \in \mathbb{E}\ \forall M_2' \in \mathbb{M}_2\ [M_1[E'\rangle M_2' \Rightarrow \exists (E, \pi) \in e((\mathbb{M}_1, \mathbb{E}, \mathbb{M}_2))\ [M_1[E\rangle \varrho^{\pi}(M_2) \wedge \exists \varphi \in \Phi_1\ [\ eideq(E', \varrho^{\varphi}(E))\ ]]]$.*

CNA state spaces contain information about reachable states and events (up to renaming) and also about their concrete interconnection. This allows us to find out how particular instances are manipulated by events. Consequently, it is possible to obtain a full state space from its CNA-variant and the set of all renaming permutations. These are the contents of the below proposition. Note that using the set of all renaming permutations is normally not necessary within practical verification tasks where concrete names of instances are not important.

Proposition 2. *Let us have an object-oriented Petri net OOPN with its CNA state space $CNAStSp$ and its set of renaming permutations Π. Then it is possible to reconstruct the full state space of OOPN from $CNAStSp$ and Π.*

3.5 Naming Rules Versus Name Abstraction

Let us now compare the techniques of using sophisticated naming rules and name abstraction.

We already know that sophisticated naming rules, possibly combined with partial order reduction, can remove many of the redundancies caused by assigning different names to dynamically arising instances with certain characteristic roles in the modelled system. But, it is not guaranteed that all such redundancies will be removed in this way. Moreover, sophisticated naming rules, even when combined with partial order reduction, do not usually help much against the second possible source of the naming problem, i.e. against the state space redundancies which arise when a model contains several instances that play somehow concurrent and symmetrical roles. At the same time dealing with object-oriented models that involve such a phenomenon does not seem to be an exception.

Name abstraction, on the other hand, can remove all the redundancy associated with the names of dynamically appearing and disappearing instances and thus can save more memory than the approaches based on sophisticated naming rules. This is a consequence of always ignoring all the different possibilities of identifying uninterchangeable instances

[3] We use $\circ$ to denote functional composition.

within otherwise identical states without any respect to the way how they were created and what was their role so far.

Thus, we can conclude that renaming can save more memory than sophisticated naming rules, even in the case of using partial order reduction. On the other hand, we might have to pay for using renaming in terms of the associated time complexity because testing markings to be equal up to renaming corresponds to solving the graph isomorphism problem. Fortunately, we can usually greatly decrease the time complexity using various heuristics proposed for dealing with general-purpose symmetries, which can be accommodated for the context of OOPNs. These techniques lead to renaming-insensitive state space hashing followed by heuristic tests of states to be equal up to renaming or, alternatively, to using a heuristically computed representation of OOPN states canonical w.r.t. renaming—more details about how to apply the different heuristics in the context of OOPNs and name abstraction can be found in [16].

To illustrate the above considerations, we can present some data obtained from a simple Prolog prototype of an OOPN state space generator, which we applied to several example models, such as classical dining philosophers, distributed philosophers, Russian philosophers, or different versions of simple systems with restartable counters [8, 16]. State spaces of these models obtained using the introduced sophisticated naming rules (without partial-order reduction and without trying to isolate naming of nested objects) had about 4.10^2–5.10^5 states. When we used name abstraction, the state spaces were reduced 1.1–414 times. From this, it is visible that name abstraction can lead to significant reductions in numbers of states, although there exist systems (as e.g. the system of distributed philosophers) where already the most elaborated sophisticated rules remove most of the naming redundancies. The speedup obtained via using name abstraction in our examples ranged from very small negative values up to 2.10^3.

So, although the above presented results could change to some degree when improving both of the mentioned methods (partial order reduction, isolated naming of unambiguously nested objects, better hash functions, bit chain-oriented comparison of states, better algorithms of name abstraction, and using self-symmetries) and although the predominance of name abstraction (and especially the obtained speedup) may decrease, it seems that we can make the following conclusion: The most elaborated sophisticated naming rules can be applied when dealing with systems without many system-level symmetries, especially when partial order reduction will be used. In the other cases (which do not seem to be only exceptional when applying object orientation), using name abstraction should probably be more advantageous. Moreover, the choice of renaming symmetries becomes natural if we want to exploit some other kind of symmetries, as well.

4 Using State Spaces of OOPNs

Most of the common ways of asking state space analysis or verification questions [15] can be accommodated for dealing with OOPN-based models, too. In the following, we will briefly discuss some of the problems associated with dealing with state space statistics, universal state space query languages, property labels or directives, and temporal logics in the context of OOPNs.

4.1 OOPN State Space Statistics

In the context of OOPNs, we can probably use all the general state space statistics (such as numbers of states, numbers of strongly connected components, sets of home states, etc.) without any significant changes. On the other hand, Petri net-based statistics (such as bounds of places, liveness of transitions, etc.) have to be adapted for the OOPN-specific style of structuring models.

In OOPNs, we can define overall place bounds and overall transition liveness ignoring the object-oriented structuring and virtually merging all instances of the same nets together. On the other hand, we might want to know what is the maximum number of tokens which can appear in a place of a single instance of a given net, or whether a transition is live within every instance of a certain net that can arise in a given OOPN.

We can also add some new statistics primarily connected to dealing with instances and objects in OOPNs. For example, we can be interested in whether there arise some (or only) persistent instances or transient instances of some net (i.e. instances which cannot be deleted or which will unavoidably be deleted after having been created).

Let us note that evaluating instance-oriented statistics is often much more complicated than evaluating statistics ignoring the object-oriented structuring [16]. This is because dealing with the former leads to tracking the individual life of particular instances and (due to recycling identifiers or due to name abstraction) we cannot simply interrelate two local states of instances referred to via the same identifier in two different states without examining what state space paths there are between the two states.

4.2 A Universal OOPN State Space Query Language

A universal state space query language for user-controlled traversing through state spaces of OOPNs can be inspired by any of the already proposed languages of this kind, such as the one associated with the tool Design/CPN [9]. However, a universal state space query language for the context of OOPNs has to provide some specialized tools for querying states and events of OOPNs respecting their highly specific structuring. Note that such tools can then also be used for describing terms embedded in temporal logic formulae specifying properties of systems to be verified by means of their OOPN-based models, for expressing system invariants to be checked, or for a fine-grained specification of legal termination states or progress states.

The main problem to be solved when querying states and events of OOPNs stems from the dynamism of OOPNs. We have to prepare tools for exploring properties of sets of states and events in a way which respects the fact that for sets of the just existing instances, their names and relations can be different in every encountered state and cannot be normally predicted.

Within a prototype of an OOPN state space tool, we have suggested a solution to the above problem based on two groups of functions. Firstly, we use the so-called *instance querying functions*. They allow us to initially obtain sets of the just existing instances of certain nets (or to start with the initial object). Subsequently, they make it possible to recursively derive sets of the net instances or constants that are straight or transitively referenced from the already known instances via the marking of some of their places or transitions. There also exists a function for deriving sets of the method net instances that are just running over some given objects. All these functions avoid an exact knowledge of the just existing instances by working with not in advance restricted sets of them. Secondly, there is a group of the so-called *set iterating functions* to be applied over sets of instances and constants. The functions from the two groups are intended to be combined in order to obtain the required characteristics of states. For example, we can take all the just existing instances of some net, select the ones which contain some constant in some place, and then proceed by exploring some other instances referenced from the selected ones.

Let us present an example of examining states of OOPNs. We will use a model of the system of distributed philosophers from [8]. The class describing distributed philosophers is shown in Figure 2. The whole model of the system of distributed philosophers contains yet another class whose only task is to create a group of philosophers and interconnect them into a ring via the methods `leftNb:` and `rightNb:`. Distributed philosophers differ from the classical ones in not having a shared table which could be used for exchanging

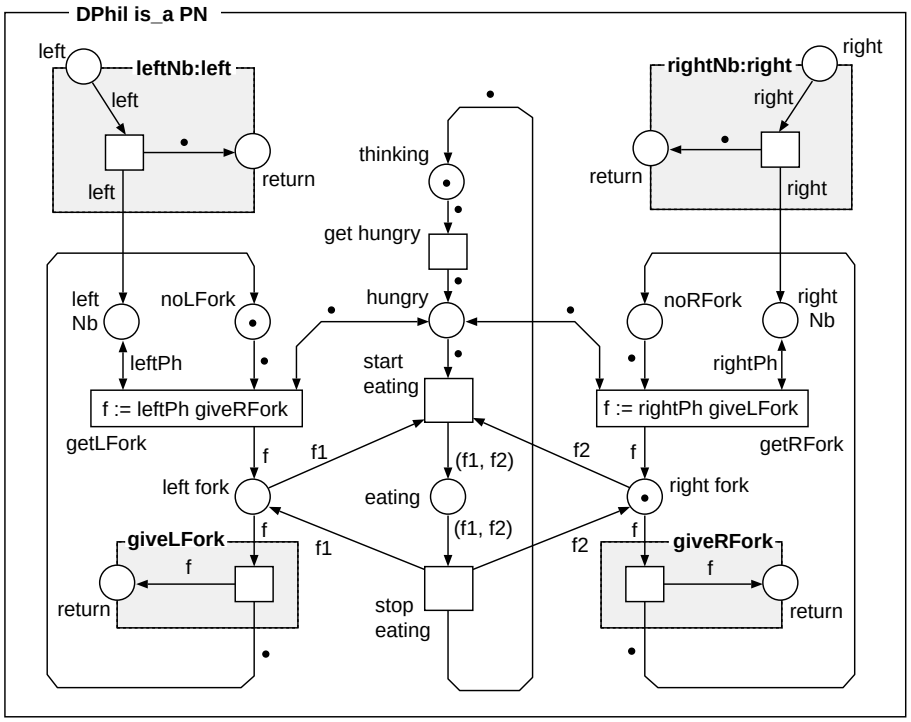

Fig. 2. The class of distributed philosophers from [8]

forks. Instead, they have to negotiate about forks via message sending using the methods `giveLFork` and `giveRFork`.

Below, we define a predicate `eating_neighbours` for deriving the set of eating philosophers having an eating neighbour from a given state of the model of distributed philosophers. We use a Prolog notation of instance querying and set iterating functions as within our prototype tool for working with state spaces of OOPNs. The current state is considered to be implicit. The query `inst(Cs,Ns,Is)` returns the set `Is` of the just existing instances belonging to the nets from the set `Ns` and running over objects belonging to the classes from `Cs`. The query `token(Is,Ps,Cs,Ms)` returns the set `Ms` of tokens belonging to the classes from `Cs` and stored in the places from `Ps` within the instances from `Is`. The query `select(S1,X,P,S2)` selects all the elements `X` from `S1` which fulfil the predicate `P` over `X`.

```
eating_neighbours(EN) :-
  inst([dPhil],[[dPhil,object]], I),
  select(I,P,(token([P],[eating],all,E1),empty(E1,false),
              token([P],[leftNb,rightNb],all,LR),
              token(LR,[eating],all,E2),empty(E2,false)), EN).
```

The predicate `eating_neighbours` can be used to check the correctness of the proposed system which should not allow two neighbours to eat at the same time. Thus the predicate should always hold for the empty set, only. This can be checked by a suitable state space query which evaluates the predicate `eating_neighbours` over every state and collects the states where it holds for a non-empty set. A more abstract approach would be checking the validity of the CTL formula AG `eating_neighbours`$(\emptyset)$.

4.3 Property Labels and Directives in OOPNs

Let us start with assertions and invariants, which are probably the simplest (yet quite useful) kind of property directives. In the context of OOPNs, *assertions* can have a form of special guards which do not restrict the firability of transitions or ports, but once a transition or a port with an associated assertion is being fired the assertion must hold (otherwise an error message is generated). *Invariants* of OOPN-based models can be expressed in the common way as conditions over states that are supposed to hold in every reachable state (if this is not the case, an error is announced). Particularly, OOPN state invariants can be specified using the above mentioned set of functions for querying states of OOPNs.

In order to allow illegal termination states of OOPN-based models to be identified and announced within generating a state space, we can introduce a special kind of labels for declaring the so-called *end-state places*. A terminal state will then be considered legal if and only if all the instances existing in that state have some tokens in their end-state places. Additionally, we may require that such a state must not contain instances of the nets that do not involve any end-state places.

When legal and illegal termination states cannot be distinguished using the above end-state place labels, we may apply general *end-state conditions*. Such conditions can be described via the functions for querying states of OOPNs mentioned in the previous subsection. An end-state condition can mark a state to be a legal termination state if it evaluates to true over such a state. This approach allows legal termination states to be recognized via complex conditions that require tokens with certain colours to be stored in certain sets of places.

In the context of OOPNs, it further seems to be useful to introduce some *instance deletion place* labels (or their generalization in the form of *instance deletion conditions*). By specifying instance deletion places in a certain net, we can say that instances of this net can be legally deleted only if the specified places are marked (otherwise an error will be announced). On the other hand, we can introduce labels indicating that all instances of certain nets are supposed to be persistent, i.e. that instances of these nets should never be deleted.

As a counterpart to the end-state place labels, we can introduce yet another kind of labels of places that will allow us to specify the so-called *progress places*. A loop encountered in a state space will then be considered legal if and only if it goes through a "progress" state which is characterized by the fact that it contains an instance whose progress places are marked. We can also introduce *progress conditions* as an analogy with the above mentioned end-state conditions.

For illustration, let us consider joining the progress label with the place `eating` from the object net of the distributed philosophers from Fig. 2. Using this progress place, we will find out that the system of distributed philosophers exhibits undesirable non-progress loops even under the strong fairness assumption. This is because the philosophers may keep exchanging forks via the transitions `getLFork` and `getRFork` and the methods `giveLFork` and `giveRFork` without enabling at least one philosopher to start eating.

Unfortunately, the above approach is not sufficient when we want to verify that each out of a set of concurrently existing instances of some net exhibits some kind of progress, and the particular instances can reach their local progress states in different states of a state space loop. To allow a specification of this kind of progress, we may introduce *instance progress places*. A loop in a (full) state space will then be considered legal if and only if it contains an instance progress state for each instance that exists throughout the entire loop. However, when verifying this kind of progress, we have to individually track the particular instances, and the associated computational complexity becomes additionally multiplied by a power of the maximum number of concurrently existing instances (w.r.t. the chosen

fairness assumptions) [5] and, in the case of CNA state spaces, we may have to unfold some of their labels.

4.4 Temporal Logic Specifications

Over state spaces of OOPN-based models, common temporal logics (such as LTL or CTL) can be applied without modifying their temporal operators (and the associated model checking algorithms). The only difficulty is how to specify atoms of formulae of such temporal logics. Nevertheless, atoms of temporal logic formulae refer to properties of particular states or events. Therefore we can describe them for example via the functions for querying states and events which we introduced within the discussion of a state space query language for the context of OOPNs.

Let us note that common temporal logics do not provide sufficient tools for tracing particular instances along state space paths in a renaming-insensitive way. For this reason, we can use the specialized temporal logic called Indexed CTL* (ICTL*) whose application over symmetrically reduced state spaces of systems consisting of many processes was described in [5]. Among other things, ICTL* introduces two new quantifiers, namely $\wedge_i$ and $\vee_i$, which act as universal and existential quantifiers over processes. There are some restrictions on how these new quantifiers can be used, but it is not necessary to discuss their details here.

When we apply ICTL* over OOPN-based models, it is obvious that $\wedge_i$ and $\vee_i$ should range over net instances and/or objects. However, the original work on ICTL* does not take into account the possibility of dynamically creating and destroying instances. To capture this phenomenon, we should define the process quantifiers in the context of OOPNs to range just over the instances which exist in the state in which we evaluate the appropriate (sub)formula. Moreover, we have to take into account that if we are examining some property of some instance over some state space path, it can happen that the instance is deleted in some moment. Since our intuition is that instances cannot be reincarnated (just only their identifiers can be reused), we shall evaluate the given formula for the given instance as though the appropriate instance deletion state was a terminal state. Consequently, if $EF\ P_i$ should hold for some instance i in some marking M, P_i must get fulfilled on some path leading from M before i is destroyed, and if $AG\ P_i$ should hold for some instance i in some marking M, P_i must hold on all paths leading from M until i is destroyed.

Fortunately, it seems to be relatively easy to accommodate the original model checking algorithm of ICTL* [5] to cope with the above mentioned modifications. Compared to a non-indexed temporal logic, the time complexity of model checking ICTL* is increased by the necessity to track the individual life of particular instances [5].

5 Conclusions

In the article, we have described how dealing with identifiers of dynamically arising and disappearing model components can worsen the state space explosion problem. We considered working with state spaces of the object-oriented Petri nets associated with the PNtalk tool, but a similar phenomenon can be encountered in other formalisms with dynamic instantiation, as well. Two possible approaches to dealing with the identifiers, namely sophisticated naming rules and name abstraction, have been described and compared. The most elaborated sophisticated naming rules can be applied when dealing with systems without many system-level symmetries, especially when partial order reduction is used. Otherwise, using name abstraction should be more advantageous. Moreover, the use of renaming symmetries becomes natural if we want to also exploit some other kind of symmetries.

Next, we have sketched how several different ways of asking analysis and verification questions can be applied in the domain of OOPNs such that their common advantages and

disadvantages are preserved. Namely, we have mentioned OOPN state space statistics, a universal OOPN state space query language, OOPN property labels and directives, and using temporal logics in the context of OOPNs. We have seen that analysis and 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. However, in both cases, modellers do not have to refer to the concrete names of the involved instances, which would be hard or impossible to manage. We have also mentioned that dealing with instance-oriented analysis or verification questions may lead to higher time and space requirements. This holds 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 notions included in the article are supposed to be exploited within formal analysis and verification on further reduced OOPN state spaces, which is one of the goals of our future research. As for further reducing OOPNs' state spaces, we intend to adapt the theory of partial order reductions for the domain of OOPNs because there is quite a lot of concurrency in models based on them.

Acknowledgement. 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".

References

1. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – Second Generation of a Java Model Checker. In *Workshop on Advances in Verification*, 2000.
2. 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.
3. M. Češka, V. Janoušek, and T. Vojnar. Generating and Using State Spaces of Object-Oriented Petri Nets. *International Journal of Computer Systems Science and Engineering*, 16(3), May 2001. A revised version of [4].
4. 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, 21st International Conference on Application and Theory of Petri Nets – ICATPN 2000*, DAIMI PB-548, pages 35–54, Aarhus, Denmark, 2000. Department of Computer Science, University of Aarhus.
5. E.A. Emerson and A.P. Sistla. Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, July 1997.
6. G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
7. C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Journal of Formal Methods in System Design*, 9(1/2):41–76, 1996.
8. V. Janoušek. *Modelling Objects by Petri Nets*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering and Computer Science, Technical University of Brno, Czech Republic, 1998. (In Czech).
9. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 2: Analysis Methods*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1994.
10. C.A. Lakos. Object Oriented Modelling with Object Petri Nets, 1999. A paper to be published in Advances in Petri Nets.

11. D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. *Journal of Formal Methods in System Design*, 8 (1):39–64, 1996. Also appeared in 6th International Conference on Computer Aided Verification 1994, Stanford CA, USA, LNCS 818, Springer-Verlag, 377-390.
12. C. Sibertin-Blanc. Cooperative Nets. In R. Valette, editor, *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 471–490, Zaragoza, Spain, June 1994. Springer-Verlag.
13. A.P. Sistla, V. Gyuris, and E.A. Emerson. SMC: A Symmetry based Model Checker for Verification of Safety and Liveness Properties, 1999. An extended version of the paper [14].
14. A.P. Sistla, L. Miliades, and V. Gyuris. SMC: A Symmetry Based Model Checker for Verification of Liveness Properties. In O. Grumberg, editor, *Proceedings of 9th International Conference on Computer Aided Verification, CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 464–467, Haifa, Israel, 1997. Springer-Verlag.
15. 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.
16. T. Vojnar. *Towards Formal Analysis and Verification over State Spaces of Object-Oriented Petri Nets*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering and Computer Science, Brno University of Technology, Czech Republic, April 2001.