Analysis and Verification Queries over 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. In literature there have been proposed many different ways of asking formal analysis or verification questions to be answered over state spaces of various kinds of models. In this paper, we discuss how these methods can be accommodated for the case when the systems being examined are modelled by object-oriented Petri nets (OOPNs). We deal with the OOPNs associated with the PNtalk language and tool.

1 Introduction

The PNtalk language and tool based on a certain kind of object-oriented Petri nets (OOPNs) have been developed at the Brno University of Technology [6] in order to support modelling, investigating, and prototyping concurrent and distributed object-oriented systems. PNtalk supports intuitive modelling of all the key features of such systems, including object-orientedness, message sending, parallelism, and synchronisation. There has already been implemented a tool for simulating systems described by OOPNs [1]. Currently we are working both on an implementation of a tool which should allow us to run OOPN-based prototypes in a truly distributed way as well as on state spaces-based methods for formally analysing and verifying properties of systems modelled by OOPNs.

Formal analysis and verification 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 [8]. 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.

In [2, 10], we have carefully examined some special issues that turn out to be related to generating state spaces of OOPNs (as it is also briefly mentioned below). This paper then provides a more detailed discussion of using state spaces of OOPNs. Particularly, we examine how the various existing ways of asking analysis and verification questions can be accommodated for the case when they are to be answered over state spaces of OOPN-based models.

In the following, we firstly introduce the main concepts of OOPNs and PNtalk. Next, we present a brief note about state spaces of OOPNs. Subsequently, we proceed to a discussion of asking analysis and verification questions to be answered over these state spaces. We finish the paper by some concluding remarks. Due to space limitations, the paper is written in an informal way—a more formal presentation can be found in [10].

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 [6].

In the following, we explain the main principles of the structure and behaviour of OOPNs. A deeper introduction to the OOPN formalism can be found in [1] and the formal definition of OOPNs in [10].

As an example illustrating the notation of OOPNs, we present a model of the system of distributed philosophers from [6]. The class `DPhil` describing distributed philosophers is shown in Fig. 1. Distributed philosophers differ from the classical ones in not having a shared table which could be used for exchanging forks. Instead, they have to negotiate about forks via message sending using the methods `giveLFork` and `giveRFork`. A run of the system of distributed philosophers is to be initiated by creating an object of the class `DDinner` (Fig. 2). This object recursively creates (by means of the method `makePh:from:to:`) a ring of philosophers interconnected by references established via the methods `leftNb:` and `rightNb:`.

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.

An OOPN *class* (as e.g. `DPhil`) 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 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). Object nets describe what data particular objects encapsulate (as e.g. references to neighbours stored in `leftNb` and `rightNb` of the object net of `DPhil`) and what activities the objects may exhibit on their own (such as the possibility to get hungry, to start and stop eating, or to get a left or a right fork).

Method nets resemble object nets but, additionally, each of them has a set of parameter places and a return place. Method nets can access places of the appropriate object nets, which allows running methods to modify the states of the

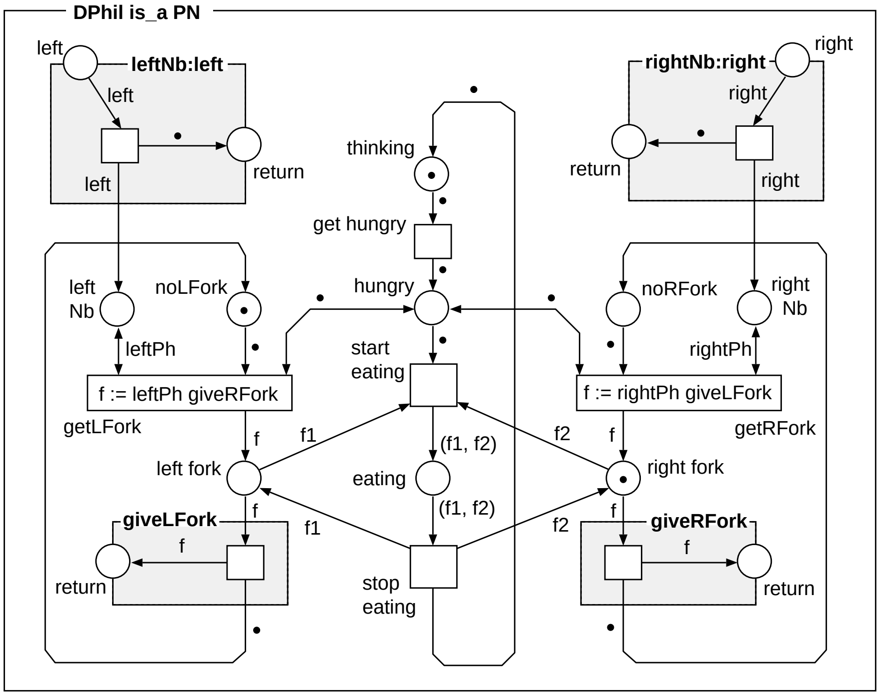

Fig. 1. The class of distributed philosophers

objects which they are running in. Method nets (as e.g. `leftNb:` or `giveLFork`) specify how objects asynchronously respond to received messages.

Synchronous ports are special transitions that cannot fire alone but only dynamically fused to some regular transitions. These transitions (possibly indirectly) activate the ports via message sending from their guards. Each port has a set of conditions, preconditions, and postconditions over places of the appropriate object net, a guard, and a set of parameters. Parameters of an activated port s can be bound to constants or unified with variables from the level of the transition or port that activated s. Synchronous ports (not used in our example) allow us 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 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 of c. Each *net instance* entails its identifier and a marking of its places and transitions. A *marking of a place* is a multiset of tokens coloured by some elements of the universe. A *marking of a transition* t is a set of records about methods invoked from t and not yet terminated.

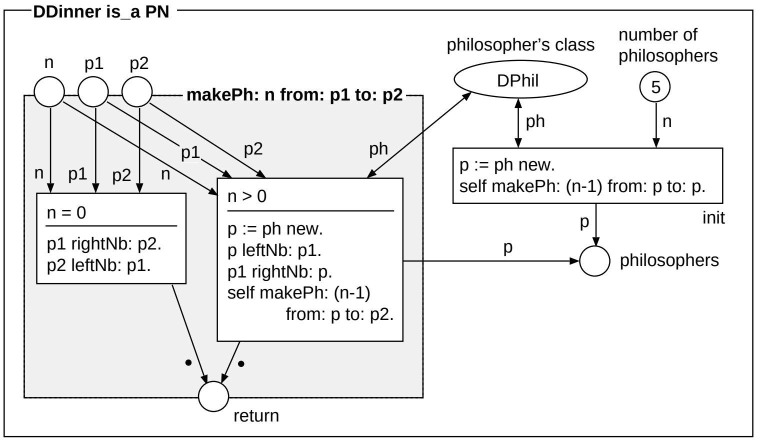

Fig. 2. The initial class of the system of distributed philosophers

For a given OOPN, its *initial marking* represents a single, initially marked object net instance from the initial class. A change of a marking of an OOPN is the 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` – a method net instance termination.

Firing an `A` event means removing and/or adding some tokens from/to the marking of certain places according to the arcs of the involved transition and synchronous ports and according to the appropriate binding. An `N` event differs from an `A` event by additionally creating and initializing a new object. An `F` event starts a new method, initializes it, and puts arguments into its parameter places. The firing transition is marked by a reference to the new method net instance and its output arcs are ignored. A `J` event retrieves a result token from the return place of a method net instance, deletes the instance and the transition marking element referencing it, and performs the output part of the appropriate transition. Garbage collection is a part of every event.

3 The Notion of State Spaces of OOPNs

In [2, 10], we have examined some special issues related to generating state spaces of OOPNs. We have especially discussed the problem of state space explosion due to generating many (semantically redundant) states differing only in the names of the involved net instances, where the names allow us to separate the instances and to express references among them, but they cannot significantly influence the future behaviour of the system being explored. We have examined sources, consequences, as well as possible solutions of this problem, which we call the *naming problem*. Although the naming problem is not exclusively specific to OOPNs, we have argued that it manifests itself in a quite severe way in the area of OOPNs (and similar formalisms), and so solving it appears to be a necessity here rather than an option.

Inspired by the methods listed in [2, 10], we have proposed and compared two methods for dealing with the naming problem in the context of OOPNs, namely *sophisticated naming rules* and *name abstraction*. Sophisticated rules for assigning names to newly arising instances attempt to decrease the degree of nondeterminism potentially present in the management of identifiers of dynamically arising and disappearing instances and thus to decrease the number of reachable states. Name abstraction is a fully transparent application of the concept of symmetrically reduced state spaces to solving the naming problem taking into account some issues specific to OOPNs.

When using name abstraction, we replace full state spaces of OOPNs by (complete) name abstracted state spaces whose states represent classes of OOPN markings that are equal up to renaming. Similarly, transitions of such state spaces represent classes of renaming equivalent OOPN events.

4 Using State Spaces of OOPNs

In literature there have already been introduced many different kinds of *query* or *specification languages* that can be used for stating analysis or verification questions within state spaces-based formal analysis or verification of concurrent systems (see e.g. [7, 8, 4, 5, 9, 3]). The different ways of asking analysis or verification questions have different pros and cons and may be advantageous in different situations. In the following, we will discuss how they can be adapted (without affecting their common advantages, of course) for the case when the systems under investigation are modelled by OOPNs. Namely, we will consider: (1) different kinds of state space statistics, (2) versatile state space query languages for user-controlled traversing through state spaces, (3) instrumenting models and using property directives, labels, or automata, and (4) using high level specification languages (such as temporal logics).

Below, we will take into account that we can be interested not only in global properties of modelled systems, but also in local properties related to the particular instances involved. We will also consider the fact that it is desirable (as we will see) to be able to avoid referring to the concrete names of instances when specifying properties to be validated.

In all the considerations, we will bear in mind that it should be possible to answer questions expressed in any of the described ways by means of examining (possibly name-abstracted) state spaces of the appropriate OOPNs. We will take care so that standard algorithms for checking different kinds of properties could be used in the new setting with as few as possible changes.

Still before we concentrate on the particular ways of asking analysis and verification questions over OOPN-based models and their state spaces, we will briefly discuss how we understand net instances to behave in time. This is necessary for a correct understanding of the spirit of the various instance-oriented properties which we will introduce in the following.

4.1 Instances and Their Behaviour in Time

From the point of view of examining temporal properties of systems, it is suitable to understand instances as entities with a unique identity and with a certain behaviour in time. In the following, the identity of an instance will not be considered the same thing as a net instance identifier that can be used for (temporarily) referring to this instance in a model. We will suppose every net to always have infinitely many instances. However, at a given point in time most of these instances will be lifeless and inaccessible. At the beginning of a run of an OOPN every net instance will be in the same special local state, which we can call a "prenatal" state. Creating a new net instance in some OOPN will then be viewed as selecting an arbitrary instance in the prenatal state, assigning a currently free net instance identifier to it (we consider the true identity of an instance to be invisible), and setting the local state of the instance to the appropriate initial marking. Similarly, deleting an instance will be interpreted as releasing the corresponding identifier (enabling its future reuse) and changing the local state of the deleted instance to a special "postmortem" state, which the instance cannot leave any more and which is the same for all deleted instances.

Formally, if we want to examine a behaviour of some (generally understood) instance in time, we deal with a sequence of equally identified OOPN net instances in a sequence of markings within a state space path such that an instance with the given identifier is present in every marking of the given sequence [10].

4.2 Statistics of State Spaces of OOPNs

In the context of OOPNs, we can use probably 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 (as e.g. bounds of places or liveness of transitions) have to be accommodated for the OOPN-specific style of structuring models. We can also add some new statistics primarily connected to dealing with instances and objects in OOPNs.

Statistics Primarily Based on Instances. The basic statistics that can be related to dealing with instances and objects in OOPNs include maximum numbers of concurrently existing instances of various nets, maximum numbers of instances of particular method nets running within a single object, and so on. Another interesting measure can be whether there arise some persistent, transient, and/or perspective instances of a certain net, or—more precisely—how many of such instances can be found in a single reachable state. We call an instance involved in some marking a *persistent instance* if it cannot be deleted in any possible future of the marking. On the other hand, we call it a *transient instance* if it is eventually deleted in every possible future. Finally, we call it a *perspective instance* if it can, but need not, be deleted. Since the active life of an instance always begins with its creation, we can also introduce the notion of an event creating a persistent, transient, or perspective instance, which allows us to relate persistence, transientness, and perspectivness to the whole active life of instances arising in a running OOPN.

To present some examples, we can say that there can be created only persistent instances of the object net of the class of distributed philosophers in the model from Figures 1 and 2. Moreover, if we restrict ourselves to (at least) weakly fair behaviours, there can arise only transient instances of the method nets `makePh:from:to:`, `leftNb:`, and `rightNb:` that initialize the system of distributed philosophers. This means that, under the weak fairness assumption, the initialization of the system of distributed philosophers always terminates.

Computing maximum numbers of concurrently existing instances of particular nets or maximum numbers of method net instances running within a single object is relatively easy. It suffers from the same problem as most other comprehensive state space statistics, only. Namely, it gathers information from all over a state space, which complicates the possibility of reducing the state space.

Checking instances to be persistent, perspective, or transient is generally more complicated. This is because we have to examine the individual life of particular instances. When doing this, we have to take into account that the same identifier can be used to refer to different instances in different states. Therefore 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. Moreover, in the case of using name-abstracted state spaces, we have to cope with the fact that when going from one state into another, the instance which we are interested in can be referred to by another identifier in the target state than in the source state.

For tracking the individual life of particular instances, we can—more or less explicitly—exploit the approach used within model checking the temporal logic called Indexed CTL* (ICTL*) over symmetrically reduced state spaces [3]. (We will mention ICTL* in more detail later on.) The principle of this approach is that we separately track the particular instances through the so-called "indexed state spaces". States of such state spaces have the form of tuples (s, id) consisting of a state s of the appropriate base state space and of an identifier id that can be used to access the local state of some instance existing in s. Transitions of indexed state spaces not based on name-abstracted base state spaces interconnect (s_1, id) with (s_2, id) if there is a transition from s_1 to s_2 and the given instance is not deleted within this transition. When using name abstraction, we have to further take into account that id can change to some id' in s_2 as it is illustrated in Fig. 3 (for more details see [3, 10]). Obviously, it is then easy to examine the behaviour of any instance once we know how it is identifed in some state.

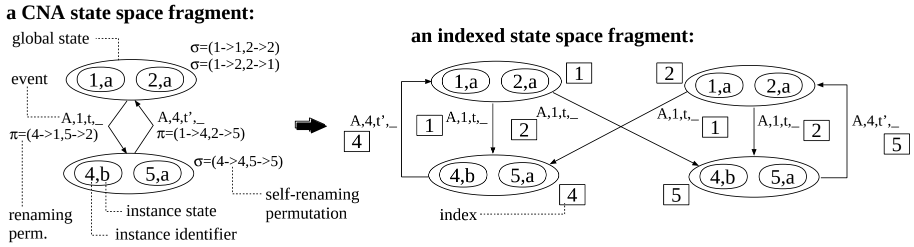

Fig. 3. The principle of indexing complete name-abstracted (CNA) state spaces [10]

To check whether some identifier id refers to a persistent, perspective, or transient instance within some marking M, to check whether there can arise some persistent, perspective, or transient instances, or to check whether there can arise only perspective or transient instances of some net, we can use the later mentioned modification of ICTL* (and its model checking algorithm) or certain specialized methods based on series of depth first searches over the appropriate indexed state spaces (or their graphs of strongly connected components) as proposed in [10]. The time complexity of such techniques is usually linear in the maximum number of concurrently existing instances of the net being examined and in the size of the appropriate state space (with unfolded transition labels and self-symmetries in the case of complete name-abstracted state spaces). The situation complicates even more when we intend to apply some fairness assumptions, which is again close to the problem of model checking ICTL* formulae. On the other hand, let us add that checking whether all instances of some net arising in some OOPN are persistent is easier than the other cases—it is enough to check that no instance of the appropriate net can be deleted.

Bounds of Places in OOPNs. Due to the object-oriented style of structuring OOPN-based models, we can distinguish *instance place bounds*, *object place bounds*, and *overall place bounds*. In the case of instance place bounds, we examine markings of particular places separately for each instance of the appropriate net. In the case of object place bounds, we summarize markings of the same places of all the instances of the appropriate net running within a single object. Finally, in the case of overall place bounds, we entirely ignore the structuring of running models into objects and instances.

Furthermore, in OOPN marking element and multiset place bounds[1] [7], we should avoid using instance identifiers because their concrete values are normally not important and hard (or impossible) to predict. Dealing with concrete instance names would especially be harmful in the context of name-abstracted state spaces where it would force us to (at least partially) unfold the appropriate renaming equivalence classes. To avoid dealing with the concrete identifiers of

[1] A marking element place bound can inform us about how many tokens with certain colours can appear in certain places. An upper/lower multiset set place bound is the smallest/largest multiset M such that every reachable marking of the given place is a sub-multiset/super-multiset of M.

instances, we should define marking element and multiset place bounds w.r.t. some (default or user-defined) function that could transform the instance identifiers used within some marking to some other renaming-insensitive values. For defining such a function, the users could use, for example, the set of specialised functions for querying OOPN states and events which we will describe later on.

Let us now present several examples. The best upper integer instance bound of the place `leftNb` from the object net of the class `DPhil` from Fig. 1 is 1, which shows that each philosopher knows about at most one left neighbour at any time. If we represent instances by their types, the best upper multiset instance bound of the place `philosophers` from the object net of the class `DDinner` is `5‘DPhil`, which corresponds to the number of philosophers intended to be created in our system of distributed philosophers.

OOPN Transition-Oriented Properties. We will now have a look at the notions of dead and live OOPN transitions, which represent two out of the most commonly used Petri net transition-oriented properties. Due to the object-oriented style of structuring OOPN-based models, we can distinguish transitions to be overall, instance, or object dead or live.

The notion of overall dead and live OOPN transitions can be defined in a relatively straightforward way ignoring the object-oriented structuring. We say that a transition is *overall dead* in some net if there is no reachable marking from which the transition could be fired in an arbitrary way within an arbitrary instance of the given net. Similarly, we consider a transition to be *overall live* in some net if from every reachable marking there can be reached another marking in which there is enabled some event based on the given transition within an instance of the given net.

A more advanced notion of transitions to be dead or live can be obtained when we define transitions to be dead or live within particular net instances. Namely, we say that a transition t from a net n is *instance dead* if there can be created an instance of n such that t can never be fired within this instance. Next, we consider a transition t from a net n to be *weakly instance live* if there cannot be created an instance of n which could get (without being deleted) into a state from which it would be impossible to eventually fire t in an arbitrary way within this instance. Finally, we say that a transition t from a net n is *strongly instance live* if instances of n cannot be deleted once they are created and it is always possible to eventually fire t within any of them.

To illustrate the above introduced notions, we can say that the transitions `start_eating` and `stop_eating` from the object net of the class `DPhil` from Fig. 1 are strongly instance live within the appropriate system. The transition `init` from the object net of the class `DDinner` from Fig. 2 is not weakly instance live nor instance dead.

We can add that we can use the standard mechanisms for checking transitions of coloured Petri nets to be dead or live [7] for checking OOPN transitions to be overall dead or live within certain nets. Checking transitions to be instance dead or live is much more difficult because we have to track the individual life of particular instances throughout the appropriate state spaces. For this reason, we can again (more or less explicitly) exploit the notion of indexed state spaces and/or their graphs of strongly connected components [10]. The computational complexity is then multiplied by (a multiple of) the maximum number of concurrently existing instances of the appropriate net and it is also increased by unfolding self-renaming symmetries and transition labels of complete name-abstracted state spaces.

4.3 Querying States Spaces of OOPNs

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 [7, 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 sets of the just existing instances, their names and relations can be different in every encountered state and cannot be normally predicted.

In a prototype of an OOPN state space tool, we have introduced two groups of functions to solve the above problem. Firstly, we use the so-called *instance querying functions* that allow us to initially obtain sets of the just existing instances of certain nets (or to start with the initial object) and to further recursively derive sets of the net instances or constants which are directly or transitively referenced from the already known instances. 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. In order to obtain the required characteristics of states, instance querying functions are intended to be combined with the so-called *set iterating functions* working over sets of instances and constants. We can, for example, 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. 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 within 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 that should not allow two neighbours to eat at the same time. This can be checked by a suitable state space query which evaluates the predicate eating_neighbours over every reachable state and not succeeds in finding a state where it would hold for a non-empty set. A more abstract approach would be checking the validity of the CTL formula AG eating_neighbours($\emptyset$).

4.4 Instrumenting OOPNs

As in the case of models described by other modelling languages, OOPN-based models can be extended by some constructions that will make it possible or easier to analyze or verify certain properties of the modelled systems. Such constructions can be created by means of the normal constructs of OOPNs, possibly extended or supplemented with some additional labels, directives, etc. In OOPNs, we can use all the common kinds of property labels and directives as well as property automata. We will now have a look at how they can be applied.

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 sets of functions for querying states of OOPNs.

To allow illegal termination states of OOPNs 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 end-state place labels are not sufficient to distinguish legal and illegal termination states, we may apply general, arbitrarily complex *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.

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. 1. 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, but 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 increases in a similar way as in the case of the later mentioned model checking of ICTL* formulae.

The common principles of using *property automata* need not be modified in the context of OOPNs. It is only necessary to be able to somehow describe them and to allow them to monitor the appropriate aspects of markings and/or events of the OOPNs being examined. For this reason, we can use the normal modelling elements of OOPNs, eventually combined with some property labels or directives. Nevertheless, due to the very complex structure and behaviour of OOPNs, it seems to be advantageous to propose some simple extension of the PNtalk language for describing property automata, which holds especially for the case when the automata should be generated automatically from some high-level property specifications

4.5 High-Level Specification Languages and OOPNs

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). There only arises a problem how to specify atoms of formulae of such temporal logics. Nevertheless, atoms of temporal logic formulae refer to properties of particular states or events and thus 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, however, 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 already mentioned, specialized temporal logic called Indexed CTL* whose application over symmetrically reduced state spaces of systems consisting of many processes was described e.g. in [3]. Among all, ICTL* introduces two new quantifiers, namely $\wedge_i$ and $\vee_i$, which act as some kind of universal and existential quantifiers over processes. Note that using ICTL* is not at variance with using name abstraction, because ICTL* formulae with correct atoms only say that something is supposed to hold for some process or for all processes without any respect to their concrete identification.

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 should 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.

As an example of an ICTL* formula related to an OOPN, we present the following specification: $AG\ \wedge_i (\texttt{hungry}(i) \Rightarrow AF\ \texttt{eating}(i))$. For our system of distributed philosophers from Figures 1 and 2, this formula asserts that if an arbitrary philosopher existing in an arbitrary reachable global state is hungry, then on all paths leading from that state, the philosopher will eventually start eating. However, a verification will tell us that the formula does not hold even under the strong fairness assumption because of the already mentioned non-progress behaviour of the considered system. Note that the predicates `hungry` and `eating` can easily be defined via the functions for querying states of OOPNs.

When model checking ICTL* formulae in the context of OOPNs, we can use the original procedure from [3]. This procedure should only be slightly modified to index states with exactly the instances that are really present in them and to correctly deal with instance deletion states as described above. 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 [3]. The complexity becomes additionally polynomial (w.r.t. the chosen fairness assumption) in the maximum number of concurrently existing instances, and we have to take into account that some CNA state space labels have to be unfolded.

5 Conclusions

In the paper, we have shown how the various existing ways of asking analysis or verification queries can be applied in the area of models based on the PNtalk OOPNs. Particularly, we have presented some useful OOPN state space statistics, we have described what specific requirements a universal query language over state spaces of OOPNs should met, we have discussed possible instrumentation of OOPN-based models, and we have also mentioned some issues related to applying temporal logics in connection to OOPNs. We have pointed out that analysis and verification queries to be answered over state spaces of OOPNs can either ignore or respect the structuring of running OOPNs into instances and we have mentioned that dealing with instance-oriented properties usually leads to a higher computational complexity. We have prepared all the ways of asking analysis and verification queries in the context of OOPNs such that we could avoid undesirable references to concrete instance identifiers. The considered specification and query techniques have been applied such that we could use as standard as possible algorithms for performing the actual analysis or verification. However, we have shown that sometimes we have to use special methods to deal with dynamically appearing and disappearing net instances and objects.

In our prototype state space tool, we have made some practical experiments with some of the described methods. Particularly, we have implemented a processor of a simple state space query language over name-abstracted state spaces of OOPNs. This processor can also be used for evaluating some of the mentioned OOPN state space statistics, for dealing with assertions or invariants, or for working with simpler temporal logic formulae if they are manually transformed onto a suitable state space query. A better tool support of the specification and query techniques that we have presented here remains a subject of our further work. It is especially necessary to implement some specialized support for dealing with all the various OOPN state space statistics, for checking properties specified by all the different property labels, and for verifying properties expressed by formulae of some (indexed) temporal logic. The employed algorithms should enable (if possible) both off-line as well as on-the-fly analysis or verification.

In the future the current prototype of the tool for formal analysis and verification over OOPN-based models should be completely reimplemented, made more efficient, and interconnected with the rest of the PNtalk tool suite. Apart from a better support of the here presented specification and query techniques, we should try to further optimize the name abstraction method used in the tool (as sketched in [10]) and to also implement some other state space reduction techniques, including especially some kind of partial order reduction. An interesting possibility how to implement formal analysis and verification over OOPNs is to try to exploit some existing non-OOPN-based formal analysis or verification tool that would allow us (may be after some extension) to cope with the problems associated with state spaces of OOPNs.

Let us further note that working with OOPN-based models of systems is likely to be complemented by exploiting various other modelling languages or notations typically used within object-oriented analysis and design. The different views on modelled systems expressed by means of these languages (object diagrams, communication diagrams, etc.) can be considered an important source of properties of the modelled systems, which could be validated by means of formal analysis or verification over the appropriate OOPNs. Currently we suppose modellers to manually select the key properties and to validate them via the answers which they can obtain to suitable analytical or verification questions specified via some of the ways described above. In the future it would be interesting to examine the possibility of some automated support for such a task.

Acknowledgment. This work is a part of the research project No. CEZ:J22/98: 262200012 - "Research in Information and Control Systems" and it was also supported by the Czech Grant Agency under the contract 102/00/1017 "Modelling, Verifying, and Prototyping Distributed Applications Using Petri Nets".

References

1. M. Češka, V. Janoušek, and T. Vojnar. PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling. In F. Pichler and R. Moreno-Díaz, editors, *Computer Aided Systems Theory and Technology – EUROCAST'97*, volume 1333 of *Lecture Notes in Computer Science*, pages 591–610, Las Palmas de Gran Canaria, Spain, February 1997. Springer-Verlag.
2. M. Češka, V. Janoušek, and T. Vojnar. Generating and Exploiting State Spaces of Object-Oriented Petri Nets. In M. Pezzé and S.M. Shatz, editors, *Proceedings of Workshop on Software Engineering and Petri Nets, 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.
3. 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.
4. G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
5. C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Journal of Formal Methods in System Design*, 9(1/2):41–76, 1996.
6. 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).
7. 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.
8. 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.

9. K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD Reference Manual. Technical Report B13, Helsinki University of Technology, August 1995.
10. 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.