A PNtalk-based Model of a Cooperative Editor

Radek Kočí *
koci@dcse.fee.vutbr.cz

Tomáš Vojnar *
vojnar@dcse.fee.vutbr.cz

Abstract: The paper presents a case study of using the object-oriented Petri nets (OOPNs) associated with the language and tool PNtalk to model a non-trivial distributed software system, namely a cooperative editor of hierarchical diagrams. In the paper there is developed an object model of (a key part of) the editor, which is subsequently transformed into an OOPN. After some refinement, the obtained model should further serve for evaluating the tools being developed for simulating, prototyping, and formally analysing and verifying distributed systems by means of PNtalk OOPNs.

Keywords: formal modelling, object-oriented Petri nets, distributed systems

1 Introduction

This paper deals with the object-oriented Petri nets (OOPNs) associated with the language and tool PNtalk [2], and considers their application to modelling a cooperative editor of hierarchical diagrams. The OOPN formalism and the PNtalk language based on this formalism have been developed at DCSE TU Brno in order to support modelling, investigating, and prototyping complex concurrent and distributed software systems. PNtalk allows for intuitive modelling all the key features of such systems, which include object-orientedness, message sending, parallelism, and synchronisation.

Models described by PNtalk can be created and their behaviour simulated by means of a specialised tool [2], which we call the PNtalk system. Currently, we are working on a new version of the PNtalk system, which should allow us to run OOPN-based prototypes in a truly distributed way. Moreover, we have already done some work towards a state spaces-based formal analysis and verification of systems modelled by OOPNs [3].

This article presents a case study of using PNtalk OOPNs to model a non-trivial distributed software system. Particularly, we consider the locking subsystem of a cooperative editor for hierarchical diagrams, which has been suggested as a "benchmark" for evaluating the use of formalisms that support some kind of dynamic structuring [1, 4]. Accordingly, we intend to further use a refined version of the presented model to evaluate the capabilities of the distributed version of PNtalk and of the tools for formal analysis and verification over OOPNs.

The paper is structured as follows. We first provide a brief introduction into the notion of OOPNs and PNtalk. Secondly, we explain our view on the cooperative editor for hierarchical diagrams. Subsequently, we develop an object model of the locking subsystem of the editor and present a part of the corresponding OOPN-based model, which captures the dynamic behaviour of this system. We finish the paper by some concluding remarks about possible modifications and use of the developed model.

* Dept. of Computer Science and Engineering, TU Brno, Božetěchova 2, 612 66, Brno, Czech Republic

2 OOPNs and PNtalk

PNtalk and the associated OOPN formalism are characterized by a Smalltalk-like object orientation enriched by concurrency and polymorphic transition execution, which allow message sending, waiting for and accepting responses, creating new objects, and performing primitive computations. OOPNs are based on viewing objects as active servers that offer reentrant services to other objects. Services provided by objects as well as independent activities of particular objects are described by means of high-level Petri nets consisting of inscribed places, transitions, and arcs. A place can be inscribed by an initial marking and an initial action (allowing a creation and initialization of complex objects to be initially stored in the place). A transition can have a guard restricting its firability and an action to be performed whenever the transition is fired. Finally, arcs are inscribed by multiset expressions specifying multisets of tokens to be moved from/to certain places by the arcs associated to a transition being fired.

A model in PNtalk is built up of Petri nets organized in classes (see Fig. 1). Every class consists of (1) an object net modelling the autonomous internal activity of objects of this class, (2) a set of method nets (including constructors) describing how these objects asynchronously respond to certain messages, and (3) a set of synchronous ports that allow states of PNtalk objects to be remotely tested and/or changed in an atomic way. All method nets of a class share access to the appropriate object net, which is implemented via allowing places of the object net to be manipulated by transitions of the method nets. Consequently, instances of method nets reacting to certain messages can influence the behaviour of the corresponding object net instance. Each method net is reentrant and has parameter places and a return place. Synchronous ports are special transitions that cannot be fired alone, but only dynamically fused to some other transitions which "activate" them from their guards via message sending. The inheritance of PNtalk classes is defined via the inheritance of places and transitions of object nets, together with the inheritance of sets of method nets and synchronous ports.

A running PNtalk OOPN consists of a set of objects, which are interconnected by references bound to tokens. Using these references, the objects can communicate by means of message sending, either asynchronously via method nets or synchronously via synchronous ports (or primitive non-Petri net-based functions). In a running OOPN, each non-trivial object is an instance of some Petri net-based class and consists of an instance of its object net and of the currently running instances of its method nets, which have been created as responses to received asynchronous messages. Particular net instances are characterized by the marking of their places and transitions (referencing methods invoked from them). In different instances of the same net, the tokens representing the marking of the same place or transition can, of course, be different. Let us add that, apart from non-trivial objects, OOPNs can also deal with trivial objects, as e.g. numbers or symbols. Such objects have the form of constants over which only trivial computations corresponding to functions in the mathematical sense can be performed.

We have already mentioned that synchronous ports can be activated from guards of transitions (and recursively from guards of other synchronous ports). Guards of transitions can also involve evaluating some trivial computations. Asynchronous messages invoking methods can be sent from actions of transitions. Actions of transitions can also be used for performing trivial computations and for creating new objects. Due to polymorphism, the way of evaluating a guard of a transition or a synchronous port or an action of a transition is decided only at run-time according to the values bound to the involved untyped transition or port variables.

An example of a simple OOPN consisting of two classes is shown in Fig. 1. Objects of the class `C2` are simple counters that can count in an autonomous way up to some value and that can be synchronously or asynchronously restarted. An object from the class `C1` (supposed to be the initial class) can create three counters and let them count with the possibility of synchronously restarting them whenever their value is above `4`. There may also be performed two successful attempts to restart a counter in an asynchronous way when its value is `2` or `3`. Not successfully attempts of this kind may be repeated.

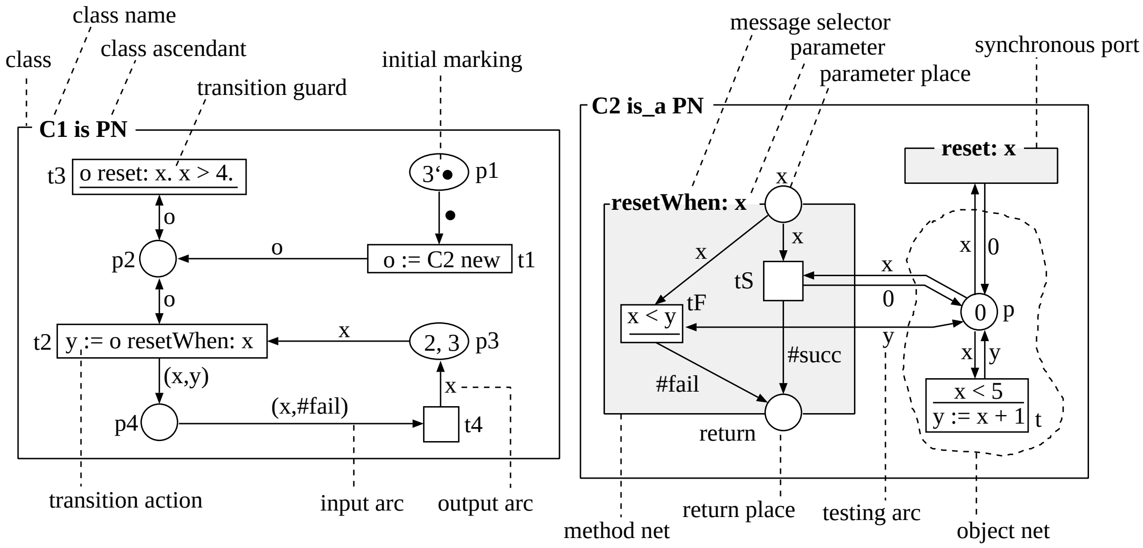

Fig. 1: An OOPN representing a simple system with restartable counters

3 The Cooperative Editor

The cooperative editor case study has been proposed to serve as a common example for formal modelling, simulation, prototyping, and analysis or verification approaches that combine a formal model of concurrency with the object-oriented paradigm [1, 4]. In the following, we briefly present our slightly modified view on how a cooperative editor for (some kind of) hierarchical diagrams should look like. We concentrate on the locking mechanisms associated with the editor.

In our view, a cooperative editor for hierarchical diagrams is a software system that should allow several users concurrently working at different workstations to cooperate on building a single diagram. Each user has to be informed in real time about actions of the other users, and the editor should provide the so-called WYSIWIS (What You See Is What I See). We do not specify any concrete type of the diagrams to be handled, but we suppose them to be hierarchically structured. Consequently, the editor must allow particular components of diagrams to be exploded into subcomponents.

We distinguish two main concepts in the cooperative editor—a diagram and a graphic component. A diagram consists of a number of components. A component can be expanded into a diagram representing the internal details of the component. On the other hand, we suppose each diagram to expand some component. In order to assure this feature, the editor maintains a single root component that expands into a root diagram which contains the roots of particular user-defined diagrams.

Before editing a diagram, a user has to access the diagram via a session. A user can establish more sessions at a time, but each session involves a single user and a single diagram. There may exist more sessions over the same diagram at the same time.

A concurrent work of several users on the same diagram can be controlled by means of some suitable editing privileges. We require a user who wants to work with some component to obtain a certain level of ownership of the component, i.e. to lock the component for a certain operation. This concept could be refined by requiring a user to enter a session with a certain level of overall privileges defining the chances of the user to view and/or edit components of the appropriate diagram, i.e. to obtain different levels of ownerships of the components. However, for the sake of simplicity, we abstract the latter concept in the following.

We define four levels of ownership of components: ownership for *viewing* that allows the owner to view attributes of a component, ownership for *encapsulation* allowing the user to explode a component into the corresponding diagram, ownership for *modification* required when modifying attributes of a component, and ownership for *deletion* that enables the owner to delete a component. Each ownership (apart from the ownership for encapsulation) is tied to a particular session and must be surrendered when the session is terminated. The ownership for encapsulation is directly related to a user and can persist between sessions. A multiple ownership of a component is allowed provided that the following rules are not broken:

- Ownership for encapsulation allows other users to lock the appropriate component for encapsulation, viewing, and modification.
- Ownership for viewing allows other users to lock the component for viewing and encapsulation.
- Ownership for modification only allows others to lock the component for encapsulation.
- Ownership for deletion requires that no-one else has any ownership of the component.

If a component is locked on several different levels of ownership (as e.g. for encapsulation and viewing), the appropriate exclusion requirements are combined. For example, if a component is locked for encapsulation and viewing, the component cannot be locked for modification because the ownership for viewing does not allow this (despite the ownership for encapsulation does). Particular ownerships may be obtained by implicitly (in the case of viewing) or explicitly (in the case of other ownerships) locking the appropriate component. Moreover, when a user creates a new component, he/she automatically owns it for deletion. An ownership may be released explicitly or at the end of a session.

4 An Object Model of the Cooperative Editor

Due to space limitations, this paper cannot cover a complete object-oriented design of the locking mechanisms of the cooperative editor introduced above. We present an object model of the system and (in the next section) a part of the corresponding OOPN-based model, only. The object model provides a static view on the involved classes and their relationships. The later described OOPN-based model integrates the essence of the object model with a model of the dynamism of the locking mechanisms (including the independent behaviour of particular objects and the way they respond to messages).

The object model of the locking subsystem of the cooperative editor is depicted in Figure 2. The model consists of the following classes:

– `RootGraphic`. An object of this class represents the root component.
– `Graphic`. Objects of the class `Graphic` play the role of components of diagrams.
– `Diagram`. An object of this class is a diagram representing a frame that encapsulates a collection of graphic components.
– `Session`. An object of the class `Session` represents a session by means of which a user works with a digram.
– `User`. Objects of this class represent particular users of the editor.

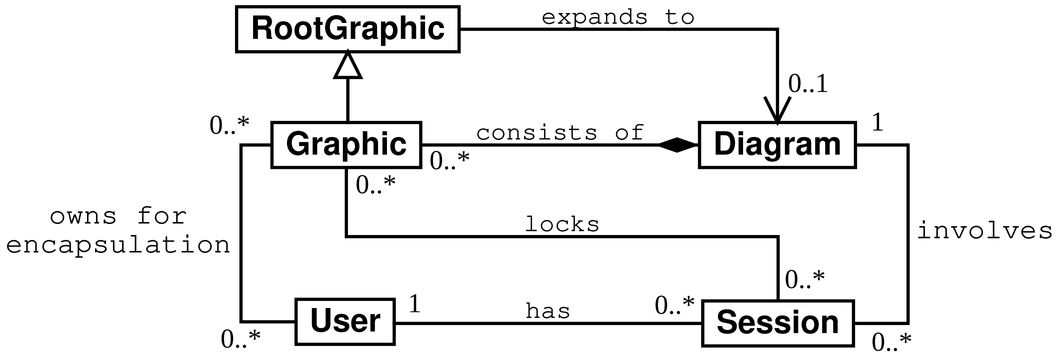

Fig. 2: The object model of the locking subsystem of the cooperative editor

We can see that there are two kinds of components in our model—common lockable components (from the class `Graphic`) and the root component (from the class `RootGraphic`) that is not lockable. The `RootGraphic`[1] does not have any editable attributes and expands into the single root diagram. Every `User` can always directly obtain a reference to the `RootGraphic`, while a reference to a `Graphic` can only be obtained from a `Diagram` being edited by the appropriate `User`.

A `User` may have several `Session`s. Each `Session` has a single `User` and involves a single `Diagram`. Within a `Session`, the corresponding `User` may lock the `Graphic`s from which the appropriate `Diagram` consists. In the case of a lock for viewing, modification, or deletion, the lock is recorded in the involved `Graphic` and `Session`. The ownership for encapsulation relates directly a `Graphic` and a `User` (and is recorded in them).

5 An OOPN-based Model of the Cooperative Editor

In this section, we present (a part) of an OOPN-based model of the locking subsystem of the cooperative editor. We concentrate on the classes `Graphic` and `Session` that play a key role in the locking mechanism. The class `Diagram` is just a framework encapsulating a set of `Graphic`s, while a `User` stores some information about a real user and models or mediates the communication between a real user and the system.

If a `User` wants to view or edit a `Diagram`, the `User` first has to create a `Session` over the `Diagram`. The session may be created by means of the constructor `user:graphic:` (Fig. 3) whose second parameter has to be the `Graphic` that is the root of the `Diagram` to be opened. The `Graphic` must be owned for encapsulation by the `User`. (The second condition is always true for the `RootGraphic`.) The ownership for encapsulation is checked via the method `expand:for:` invoked over the `Graphic`. This method returns either a

[1] We denote an object of a class A as an object A (or simply A).

reference to the appropriate `Diagram` or `nil`. The latter case causes the `Session`'s constructor to also return `nil` leading to a subsequent deletion of the not successfully created `Session` by the garbage collector.

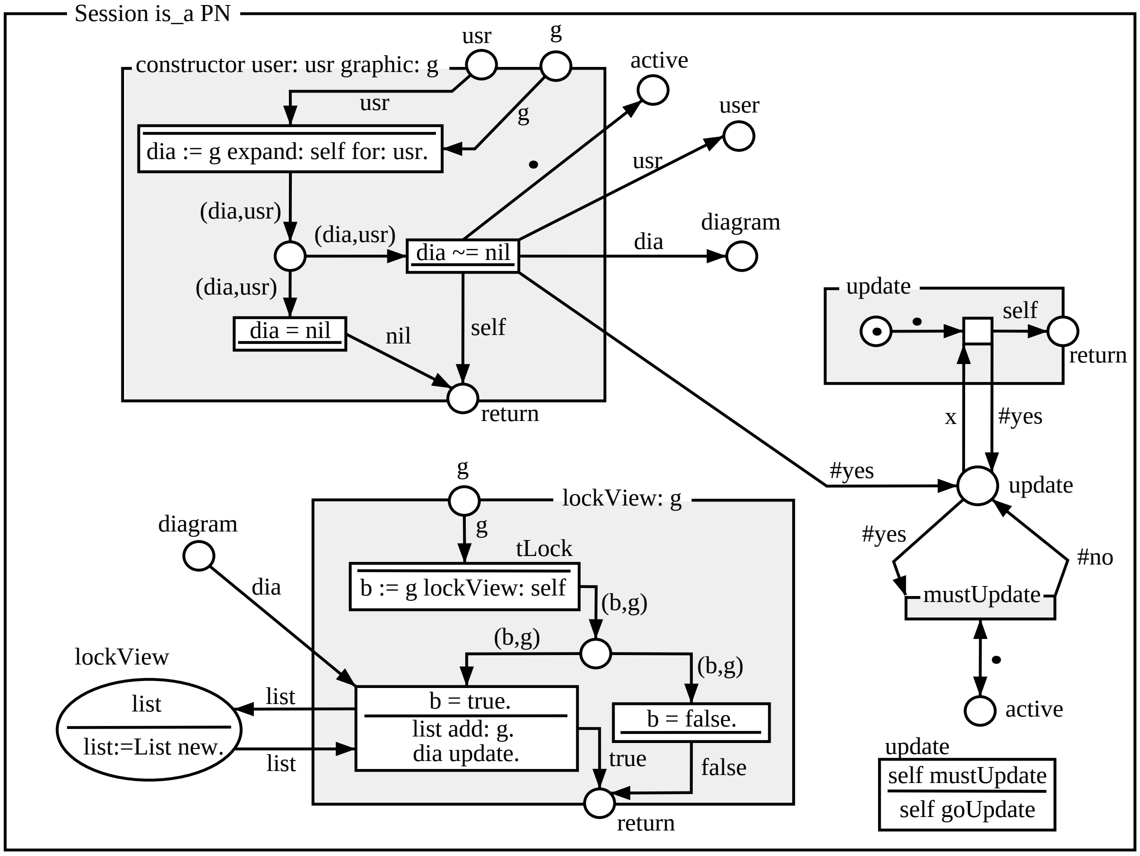

Fig. 3: A fraction of the OOPN-based class `Session`

When a `Diagram` (or its arbitrary part) is changed, it sends a message `update` to all the `Sessions` currently opened over this `Diagram`. The fact that `update` is a method allows the appropriate message to be processed in an asynchronous way and makes it easier to implement this mechanism in a distributed environment (which applies to most methods in our model). The method `update` ensures that there eventually appears a token `#yes` in the place `update` which enables the transition `update` via the synchronous port `mustUpdate` (Fig. 3). The transition `update` calls the method `goUpdate` that implements the internal details of updating a `Diagram`—the method is not presented here. Note that more requests to update may lead to a single physical update of a `Diagram`.

Let us now have a look at the core of the locking mechanisms. The process of locking is controlled by a `Session`, which may be instructed to lock a component on a certain level of ownership. The obtained lock is then stored in the `Session` and in the appropriate `Graphic`. The only exception here is the ownership for encapsulation. In this case, if the requested lock is obtained, it is stored directly into the appropriate `User`. In the following, we will only describe the process of locking a `Graphic` for viewing—the other locking procedures are similar.

A `Session` may be instructed to lock a `Graphic` for viewing by means of an invocation of its method `lockView:` (Figure 3). The transition `tLock` of this method then calls the method `lockView:` of the given `Graphic` with a reference to the involved `Session` as the argument. If the attempt to lock the `Graphic` succeeds, the `Graphic` is stored into the list of the components locked for viewing by the appropriate `Session`, which is stored in the place `lockView`. Subsequently, the message `update` is sent to the `Diagram` of the `Session`, which ensures a distribution of the `update` message to all the `Session`s currently running over this `Diagram`. If the attempt to lock the `Graphic` fails, the method `lockView:` returns `false`, and the `User` may try to lock the component later.

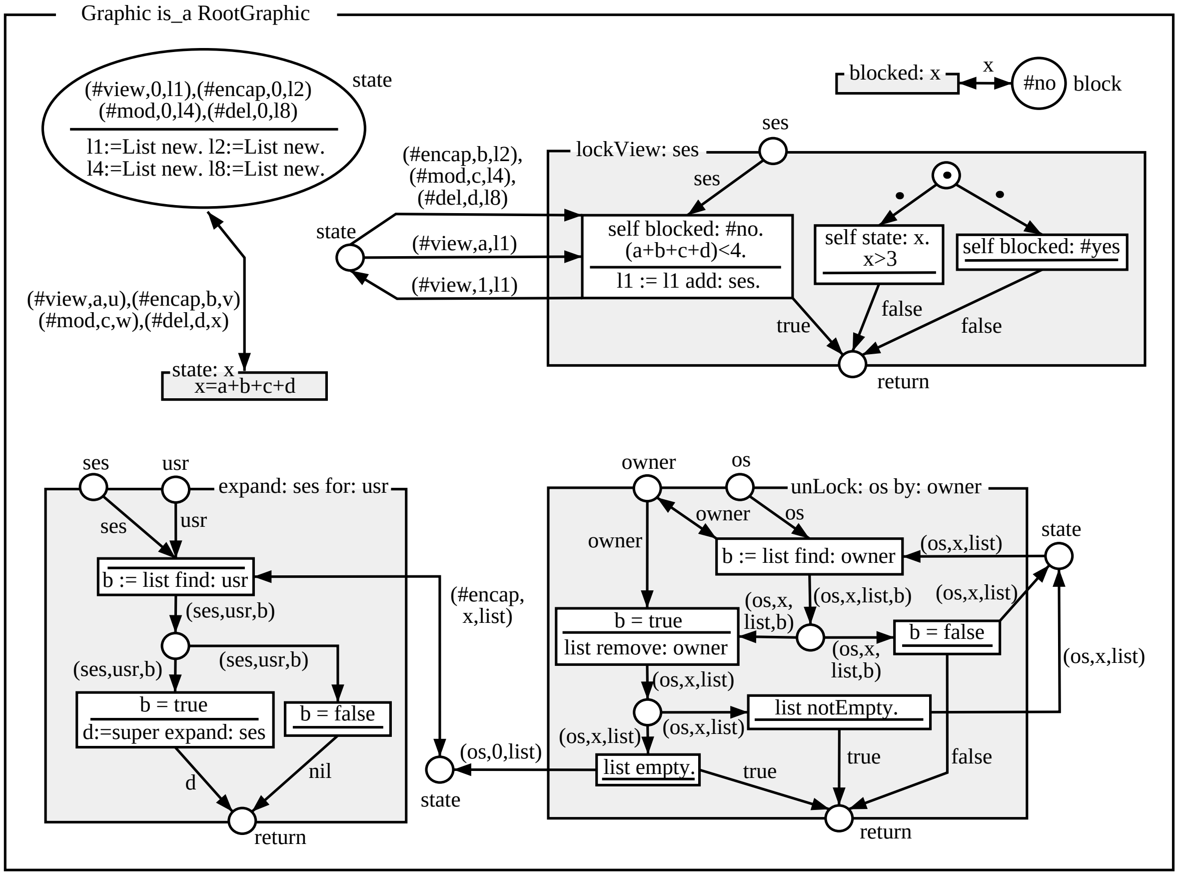

Fig. 4: A fraction of the OOPN-based class `Graphic`

The actual testing whether a component may be locked on a certain level of ownership is implemented in the class `Graphic`. The locks that are currently associated with a `Graphic` are stored in the place `state` of its object net instance. For each level of ownership, in the place `state` there is one triple consisting of a name of the level, of the corresponding characteristic number (or zero), and a list of owners. As for the characteristic numbers, particular level of ownership are associated with numerical codes that have been chosen such that they make it easier to test whether an already locked component may be further locked on some level of ownership. (For encapsulation, viewing, modification, and deletion, the characteristic numbers are `1`, `2`, `4`, and `8`, respectively.) If a `Graphic` is not locked on some level, the second item of the appropriate triple in the place `state` contains zero instead of the corresponding characteristic number.

Note that the sum of the second items of all the triples stored in the place `state` of a `Graphic` may uniquely encode any combination of locks that can be concurrently associated with the `Graphic`. Let us call this sum the “sum of locks”. Testing whether a

`Graphic` may be locked on a certain level of ownership is then based on comparing the current value of the sum of locks with the maximum value allowing a component to be locked on the given level. The maximum value may be derived from the locking rules presented in Sect. 3. In Fig. 4 there is shown the `Graphic`'s method `lockView:` which can lock a `Graphic` for viewing. (Locking on the other levels is similar.) The `Graphic` may be locked for viewing if the appropriate sum of locks is less than `4`, which means that it may be locked if it has not been locked for more than viewing and encapsulation so far (the sum of the characteristic numbers of viewing and encapsulation is `3`).

6 Conclusions

In the paper, we have briefly presented an OOPN-based model of a cooperative editor for hierarchical diagrams [1]. The model shows how the OOPNs associated with the PNtalk language and tool [2] can be applied to modelling a non-trivial concurrent software system. In the future, we intend to use the model to test the various tools which we are developing for dealing with OOPN-based models and prototypes.

We will try to refine the model to obtain a prototype of the cooperative editor. In this case, some parts of the system are likely to be implemented in other languages than only in PNtalk, which will allow us to test out the interoperability of PNtalk with other environments, such as Prolog, Smalltalk, or Java. We would also like to try out how the mobility of PNtalk objects (being currently prepared) could be used in the given area. Finally, we would like to use the model of the cooperative editor to test the methods for formal analysis and verification over state spaces of OOPN that we also developing. For this reason, some key parts of the model will have to be selected and their environment suitably abstracted to form an aggressive test bed for the part being validated. This step seems to be necessary to cope with the state space explosion problem.

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

Bibliography

1. R. Bastide, C. Lakos, and P. Palanque. A Cooperative Petri Net Editor. A Case Study proposal for the 2nd Workshop on Object-Oriented Programming and Models of Concurrency, the XVII International Conference on Applications and Theory of Petri Nets, Osaka, 1996. URL: `http://wrcm.dsi.unimi.it/PetriLab/ws96/home.html`.
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 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, ICATPN 2000*, DAIMI PB-548, pages 35–54, Aarhus, Denmark, 2000. Department of Computer Science, University of Aarhus.
4. C.A. Lakos. Object Oriented Modelling with Object Petri Nets, 1999. A paper to be published in Advances in Petri Nets.