PNtalk Modelling Experience

Milan Češka, Vladimír Janoušek, Tomáš Vojnar
Department of Computer Science and Engineering
Brno University of Technology, Božetěchova 2, 612 66 Brno, Czech Republic
Web: http://www.fee.vutbr.cz/~janousek/pntalk/pntalk.html
E-mail: {ceska,janousek,vojnar}@dcse.fee.vutbr.cz

Abstract

The paper describes a case study on applying object oriented Petri nets and the associated tool PNtalk to flexible manufacturing systems modelling. From a more general point of view, this case study is a part of a broader research on the applicability of PNtalk to modelling various complex systems in order to get some feedback and improve both the general methodology of modelling by means of object oriented Petri nets, as well as the supporting tool.

Key words: object oriented Petri nets, flexible manufacturing systems, simulation and modelling.

1. Introduction

During several recent years a formalism based on high-level Petri nets and the object-oriented paradigm has been developed at the Department of Computer Science and Engineering of the Technical University in Brno. Object-oriented Petri nets (OOPN), which combine advantages of a firm mathematical formalism with advantages stemming from the object-oriented technology, have a chance to become a tool for evaluating both correctness as well as performance of modelled systems.

Simulation is one of the ways of examining systems modelled by OOPNs and it is already supported by a prototype version of a tool called PNtalk [1]. Moreover, models created in PNtalk can be used as a basis of prototypes of the modelled systems. In such a case, some objects are likely to be implemented in Smalltalk exploiting the fact that PNtalk allows Smalltalk-based and OOPN-based objects to transparently communicate in both directions.

Together with simulation we have also started the first steps towards exploiting formal analysis and verification methods in the context of OOPNs. This approach can be considered an alternative 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 [2]. We believe that object-orientation should allow a relatively easy extraction of the subsystems to be verified together with a suitable abstraction of their surroundings.

One branch of our current research is focused on modelling and prototyping various kinds of systems by means of OOPN. This article is based on experience we have gained from a project which verified the supposed applicability of OOPNs and the associated tool PNtalk to working with more complex systems. Before we modelled only simple and artificial systems intended mainly to demonstrate features of OOPN/PNtalk. As a particular application area we have chosen the area of flexible manufacturing systems (FMS). For this purpose we established a cooperation with the Department of Production Machines, Systems, and Robotics of the Faculty of Mechanical Engineering at TU in Brno, which supplied us with some general information from the given area and allowed us to gain some experinece with a particular FMS we could model.

The model of the given FMS was created mainly in order to get some experience with FMS modelling. Nevertheless, the model can also provide a basis for further reliable considerations about the modelled system.

In the article, we first present the main ideas behind the OOPN formalism and the PNtalk language and system. Then we present a case study on applying object oriented Petri nets and the associated tool PNtalk to flexible manufacturing systems modelling. Finally, we present some future research directions concerning OOPNs.

2. Key Concepts of OOPNs

The OOPN formalism [1, 4] is characterized by a Smalltalk-based object-orientation enriched with concurrency and polymorphic transition execution, which allow message sending, waiting for and accepting responses, creating new objects, and performing primitive computations. An example demonstrating the notation of OOPNs is shown in figure 1.

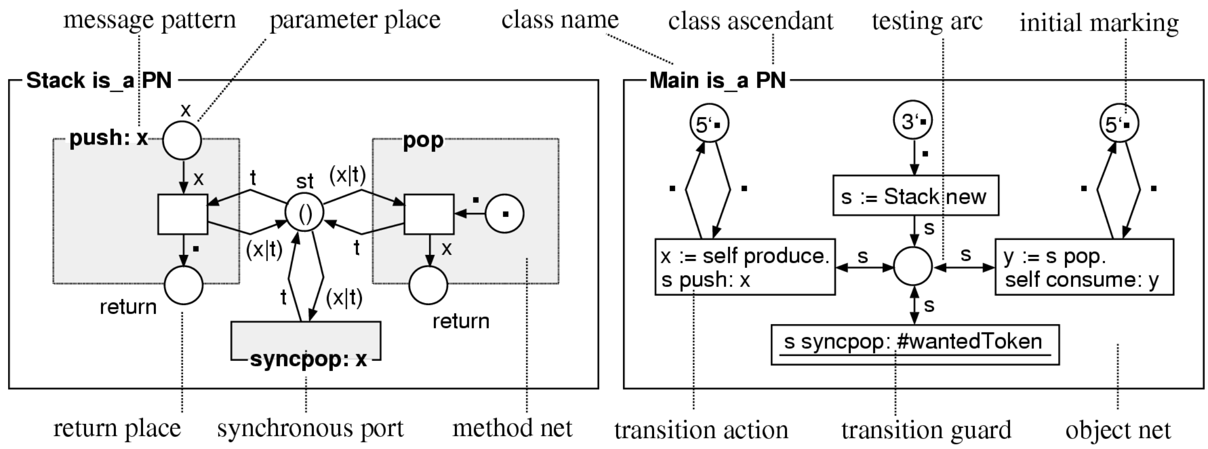

Figure 1: An OOPN example (`Main`'s methods `produce` and `consume` are not shown).

This section rephrases the basic ideas of the definition of OOPNs, however, due to space limitations, without making the description formal and complete. We explain the necessary notions only. A bit deeper introduction to the OOPN formalism can be found in [1] and the entire definition of OOPNs in [4].

2.1 The OOPN Structure

An *object-oriented Petri net* is a triple (Σ, c_0, oid_0) where Σ is a system of classes, c_0 an initial class, and oid_0 the name of the initial object from c_0.

Σ contains sets of OOPN elements which constitute classes. It comprises constants *CONST*, variables *VAR*, net elements (such as places *P* and transitions *T*), class elements (such as 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 $NET = ONET \chi MNET$ and $ID = OID \chi MID$. The universe *U* of an OOPN contains (nested) tuples of constants, classes, and object identifiers. Finally, we define $BIND = \{b \mid \text{b}: VAR \rightarrow U \}$ to be the set of all bindings of variables.

Object nets consist of places and transitions. Every place has some 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 states of objects which they are running in.

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

A *class* is 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 possible independent activities of particular objects, method nets reactions of objects to messages sent to them from outside, and ports allow for remotely testing and changing 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 mechanism applies for object net places and transitions.

2.2 The Dynamic Behaviour of OOPNs

The dynamic behaviour of OOPNs corresponds to the evolution of a system of objects. An *object* is a system of net instances which contains exactly one instance of the appropriate object net and a set of currently running instances of method nets. 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. 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*. To allow for the classical Petri net-way of manipulating markings, they are represented as multisets of token elements. In the case of a transition marking, the identifier of the invoked 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 token 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 we can say for a marking *M* that:

$$\mathrm{M} \in [(ID \times P \times U)\ \chi\ (ID \times T \times BIND)]^{\mathrm{MS}}.$$

A step from a marking of an OOPN into another marking can be described as the so-called *event*. Such an 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 $t \in T$ it is statically represented by, and (4) the binding tree b containing the bindings used on the level of the invoked transition as well as within all the synchronous ports (possibly indirectly) activated from that transition. There are four kinds of events according to the way of evaluating the action of the appropriate transition: `A` – an atomic action involving trivial computations only, `N` – a new object instantiation via the message `new`, `F` – an instantiation of a Petri-net described method, and `J` – terminating a method net instance. If an enabled event *E* occurs in a marking *M* and changes it into a marking *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*.

3. The PNtalk Language and System

Practical use of OOPNs requires a suitable computer-base tool. Therefore, the *PNtalk language and system* supporting object-oriented modeling and simulation of parallel systems [3,1] has been proposed and experimentally implemented. The PNtalk system architecture is strongly inspired by Smalltalk. It comprises a language (called PNtalk) based strictly on the OOPN formalism, its compiler, simulator, and a development environment (a browser and a debugger). The browser is intended to support creating and modifying nets within definitions of classes. The debugger allows to interactively control the simulation. Besides the interactive simulation, an automatic simulation in both simulation time as well as real time is also possible. An experimental prototype version of the PNtalk system has been embedded into the VisualWorks system (a software development environment based on Smalltalk-80). This environment makes it possible to combine Smalltalk objects with PNtalk objects, create graphical user interfaces, communicate with databases, etc. PNtalk itself allows us to draw particular nets of an OOPN, to compile them, and to process them. OOPNs can be processed either in an automatic way, when we obtain only the final results of simulation or in a step-by-step mode, which is advantageous for debugging models.

In order to test out the practical applicability of the PNtalk system in complex real-world systems, the further described model of a flexible manufacturing system has been experimentally implemented in PNtalk.

However it is ncessary to mention that the employed prototype implementation of the PNtalk language and system is based on the 1996 version of the OOPN definition. Thus, it does not fully support all the means for atomic synchronous inteactions of objects (synchronous ports) introduced in [4]. Instead, the so-called predicates allowing only non-destructive testing of object states were implemented. This has led to a bit more complicated implementation of some objects in the FMS model.

4. The Modelled Flexible Manufacturing System

From a general point of view, an FMS consists of a set of flexible machines, an automatic transport system, storage facilities, and a control system, which must be able to decide at runtime which operation has to be done on which machine [5]. Every kind of product is assigned a set of operations which it should go through. These operations can be ordered, or in the general case, partially ordered. There can be more machines capable to perform the same operation and they can have attached priorities, according to which they can be chosen for performing the given operation on the given part.

The FMS whose model we are going to discuss is a little less complicated than the mentioned general case. It is the system “CIM ER 104 Cell” produced by Eshed Robotec and installed in the laboratory of the Department of Production Machines, Systems, and Robotics of the Technical University in Brno, Czech Republic.

This system consists of four stations, a conveyor, and a central control system. In the first station there is installed a CNC lathe, the second one contains a CNC cutter. The third station includes an assembly site, a control site, and a turner. The last station contains an automatic store. All stations are equipped with a robot and a buffer with two places. Parts are stored on carriers and those are transported on pallets by the conveyor. A pallet stops every time it is going round a station and it can be loaded or unloaded by the robot of this station. A robot moves carriers among the conveyor, the buffer of its station, and machines of this station. Every station is locally controlled by its controller. The whole system is controlled by the central computer. It is supposed that there is a worker who loads semi-products into the store and unloads the final products from there.

5. A Model of the FMS Based on OOPNs

The main structure of the model reflects the physical structure of the FMS – it consists of the four stations, the conveyor, the worker, and the central control. Figure 2 depicts a partial object diagram of the FMS model – only aggregation and inheritance relations are included.

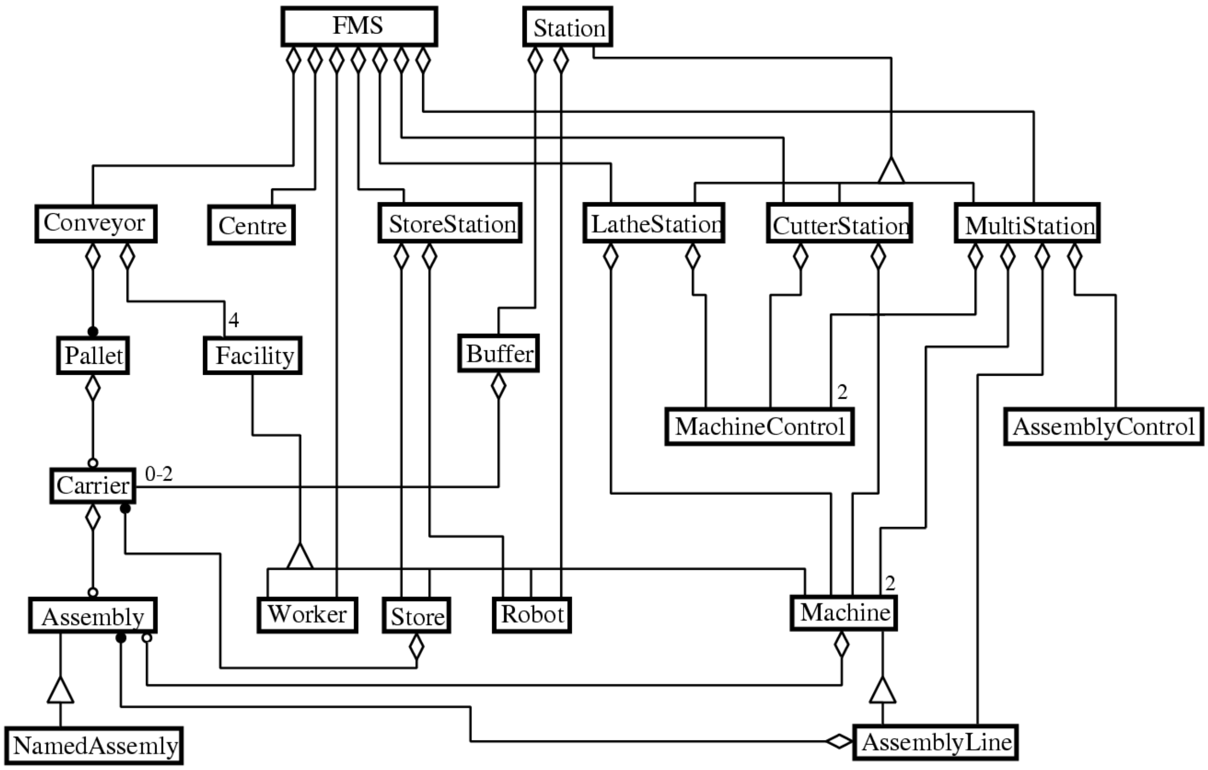

Figure 2: A (partial) object diagram of the FMS model

The three production stations are described by the classes `LatheStation`, `CutterStation`, and `MultiStation` derived from the same base class which is called `Station`. The class `Station` describes the common features of all of these stations. Namely, it implements a control strategy for exchanging carriers and assemblies between a station and the conveyor and it also provides its descendants with a robot and a buffer. A robot (an instance of the class `Robot`) is used for transporting carriers among the conveyor, the buffer of its station, and machines of this station. In the case of the multipurpose station, its robot is used for assembling parts in the assembly line, too. A two place buffer (an instance of the class `Buffer`) serves as a utility for physical storing carriers as well as a database describing its contents. The control mechanism for handling incoming and outgoing assemblies and carriers is based on requests of controls of particular machines present in a station and on the contents of the station buffer. This mechanism tries to reuse carriers and assemblies which are already in the station. Therefore it accepts a new assembly or a new carrier from the arriving pallet only, if it is needed and it is not present in the buffer. Similarly, a carrier or an assembly is sent away only, if an empty pallet arrives, and up to this moment no machine control has expressed any interest in this carrier or assembly.

Classes derived from `Station` add to the inherited properties the appropriate machines (instances of the classes `Machine` and `AssemblyLine`) and their control mechanisms (instances of the classes `MachineControl` and `AssemblyControl` – see figure 3). The task of the machine control systems is to accept requests for producing particular assemblies, to distribute requests for producing their source

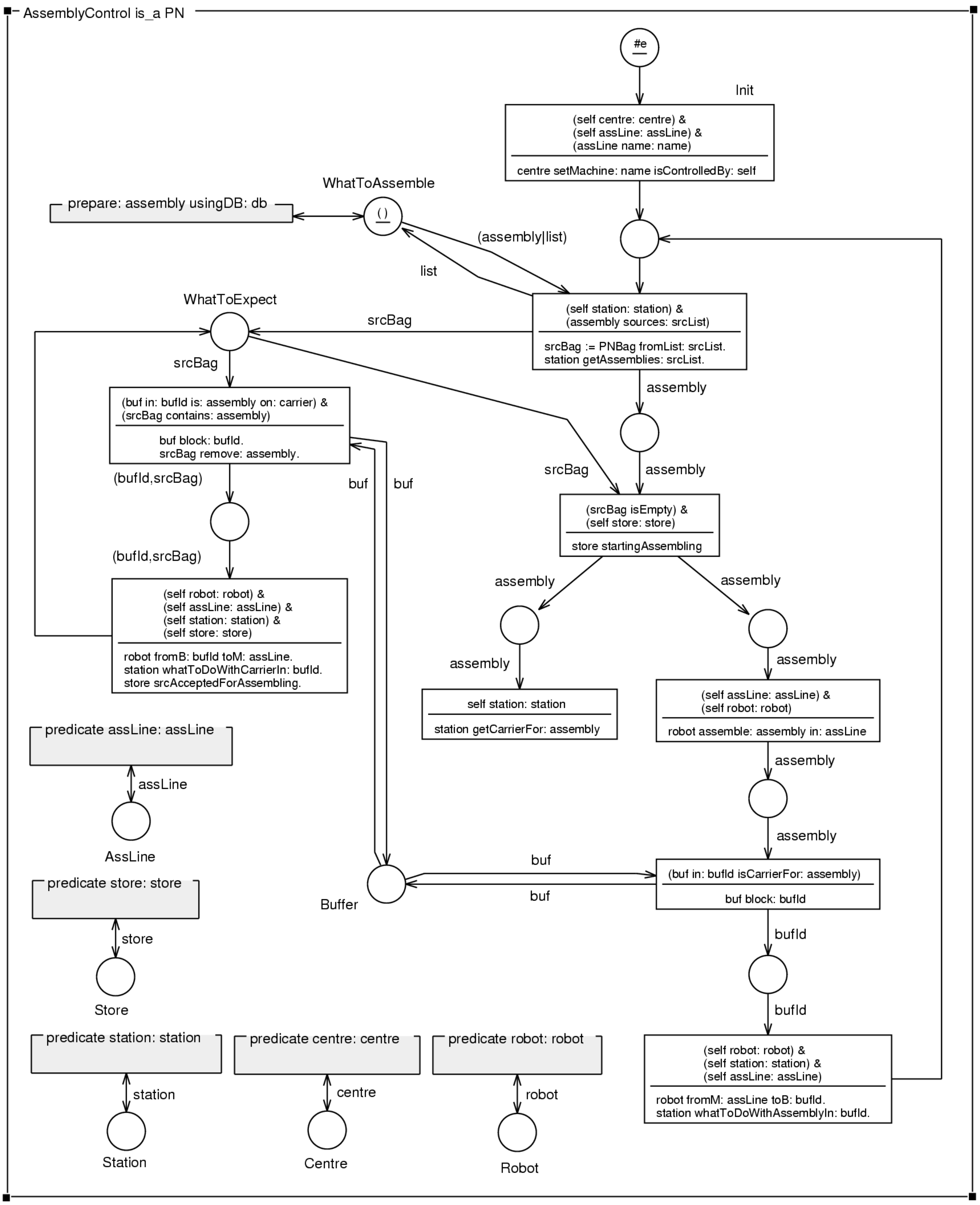

Figure 3: The object net of the class `AssemblyControl`

assemblies, to announce to station control units which assemblies and carriers are needed, and to control the movement of source assemblies into machines and of products back to the buffer. The difference between the classes `MachineControl` and `AssemblyControl` is that the assembly line control must arrange for getting more than one source assembly into the working area. The controls are included into the station classes in the form of subpages. This form has been chosen to allow sharing buffers and robots by particular machine and station control mechanisms and to allow their locking by simply removing references to them from the particular places.[1]

The fourth station containing the store is described by the class `StoreStation`, which is not a descendant of `Station`. This is because it is a little different from the other stations. `StoreStation` contains a store (an instance of the class `Store`) and a robot, but not a buffer – the buffer can be viewed as an extension (or a "cache") of the store and we have left it out to simplify the model. The class `Store` represents the physical store as well as the database describing its contents. The class `StoreStation` contains also a control mechanism for moving carriers and assemblies to and from the store. The number of movements is optimized by first trying to reuse the carriers and assemblies just going back to the store on the conveyor rather than immediately releasing new items from the store.

The task of the conveyor (an instance of the class `Conveyor`) is easy. It simply moves pallets among the terminals of the four stations. In front of every station terminal there is a queue of waiting pallets.

The worker, whose role is adding new carriers and semi-products into the store and removing final products from it, is represented by an instance of the class `Worker`. At the beginning he adds an initial number of carriers and semi-products into the store. Then he regularly adds new assemblies instead of the ones which have been consumed and removes produced final products. What is more, in the case of some critical situation in the store (something is missing, the store is full, etc.), the worker is signalled to try to solve this situation. However, his effort can be in vain and then the system can go in deadlock.

The last component of the model, we will discuss here, is the central control (an instance of the class `Centre`). The production control is based on two principles: a fixed maximum amount of work in progress and a fixed production mix. Those are implemented in such a way that the centre contains a circular list of final products, through which it keeps going launching the production of particular final products. However, it can start the production of a next product only, if it has already released all semi-products needed for producing the previous product from the store. The production of a single product is based on instructions which have the form of a tree of interconnected objects of the classes `Assembly` and `NamedAssembly`. The root of this tree is the final product we want to produce. Leaves of the tree are semi-products stored in the store. Every intermediate node represents an assembly produced by some machine. Every assembly contains information about the machine and the NC program which are used for producing it, about the carrier necessary for transporting it, and also about the source assemblies from which it is produced. So, the production of a particular product is launched in such a way, that we go through its instruction tree starting in the root and announce all the appropriate machine controls what they will have to produce and from what sources. This procedure begins by telling the store it will have to accept some final product and ends by telling the store it will have to release some semi-products. (This is, of course, done individually for every particular semi-product.) As soon as the first of the semi-products is released, the physical

---

[1] In the new PNtalk implementation being prepared, this could be implemented in a better way using synchronous ports.

process of the production begins. But, we have to take into account that only a limited number of assemblies can be present in the FMS at the same time and therefore sometimes it is necessary to wait for some time, before a next semi-product can be released. The machine controls wait until the sources arrive and then they can produce the expected assembly and send it further. If the requests for producing arrive to stations faster than they can be granted, they are organized in queues. There is a separate queue of products for every particular source assembly. Thus it can be decided what to produce according to the available source assembly and the head of the queue attached to it. The production of a single product is then finished by storing it into the store.

At the end of this section we have to stress that we are using the term *assembly* not only for products consisting of several parts, but generally for all elements which can be transported or stored on carriers and processed by all kinds of machines. Further, two physically distinct assemblies of the same type are represented by two references to the same object of the class `Assembly`. So, we do not distinguish between logical descriptions of assemblies and their physical instances. This could be necessary, if we e.g. wanted to keep some records about the quality of particular assemblies. However, we have not taken this into account.

6. Conclusions

The task of modelling the FMS has brought us some pieces of knowledge which will help us to improve the OOPN formalism, the PNtalk system, and the methodology of their usage.

It has turned out that OOPNs and PNtalk have been designed as a very general language for modelling and programming highly dynamic systems. This OOPN dynamism was not exploited to a very large extent in the case of the FMS model – its structure does not change during a run of the model. Nevertheless, the OOPN message sending mechanism proved to be relatively advantageous and natural for modelling the involved control system.

Static hierarchical structuring of the model was allowed by the adopted concept of substitution transitions [6]. However, this concept was used only inside classes `LatheStation`, `CutterStation`, and `AssemblyStation`, where controls of the particular machines were specified as subpages of the control of the whole station. This structuring mechanism was chosen because it allows simple resource allocation in the test-and-set manner by means of places shared by several nets as though they were a part of the same net. The new PNtalk implementation allows to solve this situation by means of atomic synchronous interactions of objects (using synchronous ports).

As a simple example of the methodological pieces of knowledge we have gained, we can present e.g. our experience that it is advantageous to use calling predicates via `self` from transitions to access attributes of objects rather than to use test arcs. The semantics is the same but we avoid a necessity of introducing many crossing arc (see figure 3). Moreover, a need for some more powerful synchronous interaction of objects appeared. Synchonous ports which were consequently added to the OOPN formalism should cover this need.

When developing the model of the FMS we encountered some weaknesses of PNtalk. They stem from the fact that this tool was developed mainly to illustrate problems arising in the area of the OOPN formalism [4], which was originally presented on a purely abstract level suitable for theoretical considerations only. Therefore, using PNtalk for modelling and simulation of real systems was an attempt which brought us some experience and suggestions for its future development.

In the near future we plan to improve the PNtalk language and radically rebuild the PNtalk interpreter to allow distributed simulation and realistic prototyping of large systems. Moreover, we plan to improve the PNtalk interoperability with language environments such as Smalltalk, Prolog and C++, and with middleware, such as CORBA. This will allow us to test suitability of OOPNs for modeling and rapid prototyping in more comprehensive case studies.

Acknowledgment

This work was supported by the Grant Agency of the Czech Republic under the contract GACR 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] 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.

[3] V. Janoušek. PNtalk: Object Orientation in Petri Nets. In *Proceedings of European Simulation Multicoference ESM'95*, pages 196–200, Prague, Czech Republic, 1995. Czech Technical University.

[4] V. Janoušek. *Modelling Objects by Petri Nets*. PhD thesis, Department of Computer Science and Engineering, Technical University of Brno, Czech Republic, 1998. (In Czech).

[5] M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*, Wiley Series in Parallel Computing, 1995.

[6] K. Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use, Volume 1*, Springer-Verlag, 1992.

[7] M. Češka and V. Janoušek. Object Orientation in Petri Nets. In *Object Oriented Modelling and Simulation, Proceedings of the 22nd Conference of the ASU*, pages 69–80, Clermont-Ferrand, France, September 1996. University Blaise Pascal.

[8] V. Janoušek and T. Vojnar. The PNtalk Project, 2000.
http://www.fee.vutbr.cz/~janousek/pntalk/pntalk.html and
http://www.daimi.au.dk/petrinet/tools.db/pntalk.html

[9] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut fűr Instrumentelle Mathematik, University Bonn, Germany, 1962. Available as Schriften des IIM Nr. 2. (In German).

[10] J. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice Hall, Engelwood Cliffs, New Jersey, 1981.