Discrete-Event Systems, Modeling, Simulation, Object-Oriented Petri Nets

Vladimír JANOUŠEK and Radek KOČÍ[1]

# EMBEDDING OBJECT-ORIENTED PETRI NETS INTO A DEVS-BASED SIMULATION FRAMEWORK

The importance of formal models used in system design is continually growing. Since various formalisms are suitable for different kinds of systems, their combination is valuable in description of complex systems. The paper deals with a combination of Object Oriented Petri Nets (OOPN) and DEVS formalisms. DEVS is rather static formalism using explicit component interconnections, while OOPN is a highly dynamic formalism with implicit inter-object relations. Although these formalisms are different, both are state-centered and, thus, their integration is well-feasible. The resulting framework derives benefits from both formalisms.

## 1. INTRODUCTION

In the area of software development, models have been used for a long time because of their ability of system abstraction. They allow a better quality of thinking about a developed system and, consequently, are taken as a part of system documentation. However, the relationship between models and software implementation is informal and depends on the programmer's interpretation. Furthermore, when a software system implementation changed, the model should be adapted too. It can be a complex task which may become inconsistent. Our research is focused on an application of formal models and simulation in the system development. The key idea is that models do not constitute just documentation, but are equal to code (these techniques are commonly known as *model-based design*). Models are used as an executable program valid through all the development stages including the target application. There is a lot of paradigms suitable for this approach to systems development. We are concentrating especially on DEVS and Petri nets. When creating a model, we can use each of them separately or their combination. Combination of a variety of paradigms can benefit from special features of the formalisms.

DEVS is a systems specification formalism developed by B. Zeigler [13]. It is based on general systems-theoretic concepts of state, time, input, output and hierarchy. It constitutes a basis for a theory of modeling and simulation and it can be considered as a basic platform for multiparadigm modeling and simulation. Recent research efforts around DEVS are devoted to model-based design

[1]Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 66, Brno, Czech Republic, email: {janousek, koci}@fit.vutbr.cz

and model continuity [7], as well as to interoperability and standardization in the area of modeling and simulation. On the other hand, Petri nets constitute an abstract formalism allowing a natural description of parallelism, synchronization and non-determinism. Since they deal with time on a more abstract level, they are better suited for a high-level specification of parallel systems. Nevertheless, there are many modifications of Petri nets which can also be used for simulation. We have developed a Petri net-based formalism suited for model-based design of systems. The formalism is called Object Oriented Petri Nets (OOPN) and the associated tool is called PNtalk [3]. PNtalk has been designed as a concurrent object oriented language which uses High-level Petri nets for a specification of objects. It allows for interoperability with other object oriented languages and can be used as a high-level visual programming language in a way conform with up-to-date software engineering methodologies.

Our current research is oriented towards more interactivity during systems development and a possibility to investigate the system structure and behavior by the system itself what is needed for self-development of systems. OOPN uses a coupling schema of inter-object relations based on object pointers. But, to fit our needs, we need looser relations between components. The DEVS formalism offers asynchronous communication of loosely coupled components. In order to support DEVS-based modeling and simulation of systems and experimention with self-modifiable and evolving systems, we have developed a tool called SmallDEVS. Originally, it has been developed independently of OOPN/PNtalk. Since it appeared to be relative easy to incorporate other formalisms to DEVS, we are using SmallDEVS as a component framework to which OOPN/PNtalk is embedded.

The paper is organized as follows. Firstly, we give basic information on DEVS. Then, we will define the Object oriented Petri nets formalism and will describe the OOPN dynamism. The next chapter deals with the problem of embedding OOPN into the DEVS-based framework. At the end, we discuss results and benefits of combination of OOPN and DEVS formalisms.

## 2. DEVS

DEVS is a formalism which can represent any system whose input/output behavior can be described as sequence of events. DEVS is specified as a structure

$$M = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta)$$

where

$X$ is the set of input event values,
$S$ is the set of state values,
$Y$ is the set of output event values,
$\delta_{int} : S \longrightarrow S$ is the internal transition function,
$\delta_{ext} : Q \times X \longrightarrow S$ is the external transition function, where
$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the set of total states,
$e$ is the time passed since the last transition (elapsed time),
$\lambda : S \longrightarrow Y$ is the output function,
$ta : S \longrightarrow {R^+}_{0,\infty}$ is the time advance function.

At any time, the system is in some state $s \in S$. If no external event occurs, the system is staying in state $s$ for $ta(s)$ time. If elapsed time $e$ reaches $ta(s)$, then the value of $\lambda(s)$ is propagated to the output and the system state changes to $\delta_{int}(s)$. If an external event $x \in X$ occurs on the input in time $e \leq ta(s)$, then the system changes its state to $\delta_{ext}(s, e, x)$.

This way we can describe atomic models. Atomic models can be coupled together to form a coupled model. The later model can itself be employed as a component of larger model. This way the coupled models allow to construct models hierarchically. Coupled DEVS is defined as a structure

$$CM = (X, Y, \{M_i\}, EIC, EOC, IC, select)$$

where

$X$ is the set of input event values,
$Y$ is the set of output event values,
$M_i$ is set of models of subcomponents,
$EIC \subseteq X \times \bigcup X_i$ is set of external input connections,
$EOC \subseteq \bigcup Y_i \times X$ is set of external output connections,
$IC \subseteq \bigcup Y_i \times \bigcup X_j$ is set of internal connections,
$select : 2^{\{M_i\}} - \emptyset \longrightarrow \{M_i\}$ is the tie-breaking function for simultaneous events.

Sets $S$, $X$, $Y$ are obviously specified as structured sets. It allows to use multiple variables for specification of state and we can use input and output ports for input and output events specification, as well as for coupling specification. A lot of extensions and modifications of the original DEVS has been introduced, such as parallel DEVS [4] or dynamic structure DEVS [2] and a lot of simulation frameworks has been developed. Variable structure and reflectivity are main features of SmallDEVS, a DEVS-based framework developed recently and used by our group [9].

## 3. OBJECT ORIENTED PETRI NETS

General ideas about Petri nets have been developed by C. A. Petri in early sixties. The Petri nets are powerful graphical tools having a mathematical foundation suitable for describing systems and their processes that can be termed as asynchronous parallel and distributed. There are several kinds of Petri nets. The most general ones are high-level Petri nets, e.g., Predicate-Transition nets [5], or Coloured Petri nets [11]. There ale also some formalisms derived from high-level Petri nets. We use Object Oriented Petri Nets (OOPN) [8, 10] and associated PNtalk system being developed by our research group. Object-orientation of OOPN is based on the well-known, class-based approach in Smalltalk-like style [6]. The classical kind of object-orientation is enriched by concurrency in OOPN. Each object offers services to other objects and at the same time it can perform their own independent activities. Services and autonomous activities are described by means of high-level Petri nets – services by method nets, object activities by object nets. Apart from the concurrency of particular nets, the transitions themselves represent concurrency inside nets.

### 3.1. THE OOPN STRUCTURE

An OOPN is a set of classes specified by high-level Petri nets. Formally, OOPN comprises constants $CONST$, variables $VAR$, places $P$, transitions $T$, object nets $ONET$, method nets $MNET$, synchronous ports $SYNC$, message selectors $MSG$, classes $CLASS$, object identifiers $OID$, and method net instance identifiers $MID$. We denote $NET = ONET \cup MNET$ and $ID = OID \cup MID$. For OOPN, we define universe $U$ as the set of tuples of constants, classes, and object identifiers. The set of all bindings of variables used in OOPN is then defined as $BIND = \{b \mid b : VAR \longrightarrow U\}$.

*Object nets* consist of places and transitions. Every place has its 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 (virtual) 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 autonomous activities of objects, method nets describe reactions of objects to messages sent to them from the outside, and synchronous ports allow for remotely testing states of objects and changing them in an atomic way. Classes can be specified incrementally using *inheritance*. The inherited methods and synchronous ports can be redefined and new methods and synchronous ports can be added. The same mechanism applies for object net places and transitions.

An example demonstrating the notation of OOPN is shown in Figure 1. It consists of classes C0 and C1. Class C0 contains only the object net. Class C1 contains its object net (place p and transition t), synchronous port *state:*, and methods *wait:* and *reset*.

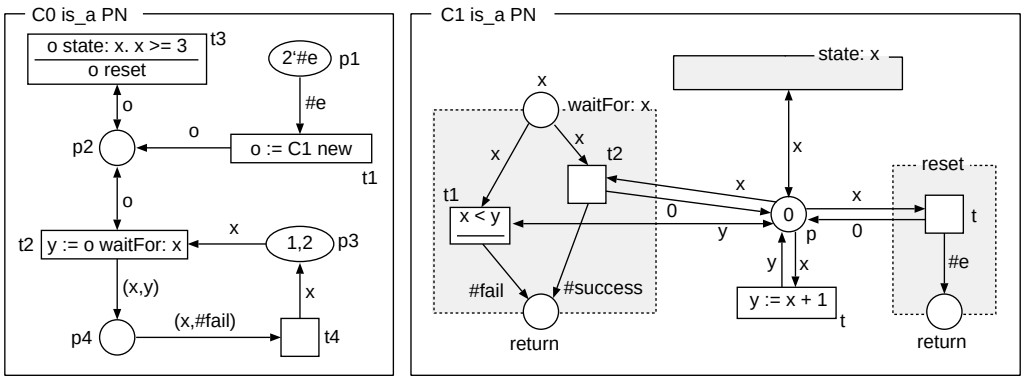


Figure 1: An OOPN example.

## 3.2. OBJECT INTERACTIONS

Object interactions in OOPN are initiated by *transitions*. Transitions in high-level Petri nets may have *guards* and *actions* (Figure 2). Transition guards are evaluated during transition firability testing. There must be no side effect. It only allows atomic decision if the transition can or cannot be executed. On the other hand, transition actions are evaluated during transition execution. There is also possible to think about non-atomic execution of transition action (start and end of action) and a possibility of a side effect (such as a message sending in OOPN).

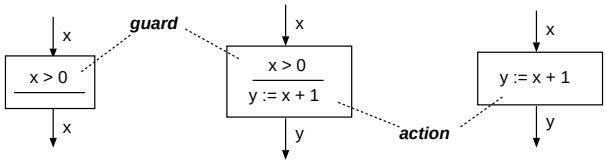


Figure 2: Transition guard and action.

In OOPN, each expression of inscription language specifies a message sending. If the receiver of the message is a primitive object (i.e. not described by an OOPN class), then the semantics is the same as in the case of the well-known high-level Petri nets. Let us now suppose a non-primitive object as a receiver. Since there is a difference in semantics of a guard and an action we can distinguish two kinds of object interactions. Both these interactions are specified as message sending: A message sending in a transition guard causes *atomic synchronous interaction*, which is realized as *synchronous ports invocation* (will be discussed later). A message sending in a transition action causes the client-server (*request-reply*) interaction which is realized by *invocation of method nets*.

*Method nets* are *dynamically instantiated* by message passing specified by *transition actions* (Figure 3). Events associated with a method net invocation are $F$ (fork, creation of a new process—an instance of the method net) and $J$ (join, terminating of the method net instance).

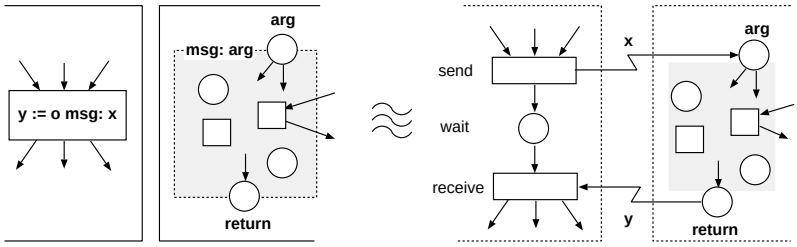


Figure 3: Client-server interaction.

*Synchronous ports* are intended for synchronous interaction of objects. The synchronous interactions (invocation of synchronous ports) are specified in transition guards as message sendings. A (sender) transition is firable only if the receiver of the message in its guard agree with it (the involved synchronous port is firable). The synchronous port is then *executed simultaneously* with the sender transition. The semantics of the synchronous interaction can be described as a transition which is a fusion of the sender transition and the synchronous port (respecting polymorphism and parameters binding) (Figure 4).

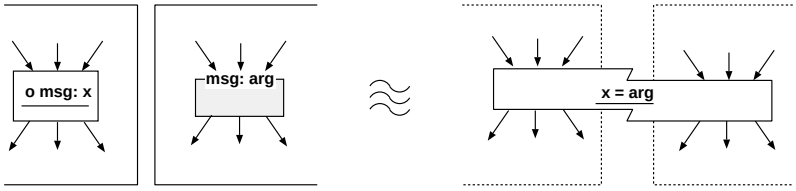


Figure 4: Atomic synchronous interaction.

## 3.3. THE DYNAMIC BEHAVIOUR OF OOPN

The dynamic behaviour of OOPN 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* of a system of net instances. To allow for the classical Petri net-way of manipulating markings, they are represented as the 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) transition 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 state $s$ that:

$$s \in [(ID \times P \times U) \cup (ID \times T \times BIND)]^{MS}.$$

A step from a state of an OOPN into another state can be described using event

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

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 $ev$ occurs in a state $s$ and changes it into a state $s'$, we call this a *step* and denote it by

$$s \; [ev\rangle \; s'.$$

Each step comprise garbage collecting—deletion of net instances which are not reachable (by references) from the initial object.

Let us have an example presented in Figure 1. The initial state of our OOPN is defined by the state of the initial object (instance of the initial class). It contains initial marking of C0's object net. Let the initial object as well as its object net instance have name $id_0$. The initial state (named $s_0$) of our OOPN can be specified by following table

| $s_0$ | | | $id_0$ |
|---|---|---|---|
| | | | $id_0$ |
| C0 | *object net* | p1 | 2'#e |
| | | p2 | empty |
| | | p3 | 1, 2 |
| | | p4 | empty |
| | | t1 | empty |
| | | t2 | empty |
| | | t3 | empty |

Suppose the following sequence of steps: $s_0\ [(N, id_0, C0.t1, ())\rangle\ s_1\ [(F, id_0, C0.t2, (x = 1, o = id_1))\rangle\ s_2\ [(F, id_0, C0.t2, (x = 2, o = id_1))\rangle\ s_3\ [(A, id_1, C1.t, (x = 0))\rangle\ s_4\ [(A, id_2, C1.waitFor : .t2, (x = 1))\rangle\ s_5$. The resulting state $s_5$ looks like follows:

| $s_5$ | | | $id_0$ | $id_1$ | | |
|---|---|---|---|---|---|---|
| | | | $id_0$ | $id_1$ | $id_2$ | $id_3$ |
| C0 | *object net* | p1 | #e | | | |
| | | p2 | $id_1$ | | | |
| | | p3 | empty | | | |
| | | p4 | empty | | | |
| | | t1 | empty | | | |
| | | t2 | ($id_2$, {(x, 1), (o, $id_1$)}), ($id_3$, {(x, 2), (o, $id_1$)}) | | | |
| | | t3 | empty | | | |
| C1 | *object net* | p | | 0 | | |
| | | t | | empty | | |
| | waitFor: | x | | | empty | 2 |
| | | return | | | #success | empty |
| | | t1 | | | empty | empty |
| | | t2 | | | empty | empty |

The system of objects contains two objects identified as $id0$ and $id1$. Marking of some places and transitions of the object $id0$ has changed. New object $id1$ contains instances of nets identified as $id1$, $id2$, and $id3$. Net instances $id2$ and $id3$ are the invocations of the method `waitFor:` created by the transition $t2$ occurrence in the object $id1$.

## 4. EMBEDDING OOPN TO DEVS

The DEVS formalism brings a hierarchical component architecture, which OOPN misses. On the other hand, OOPN/PNtalk is a powerful language based on high-level Petri Nets enabling a high-level description and dynamicity of models. There are several techniques to integrate various formalisms. The most common ones are *combination*, *mapping*, and *wrapping*. The combination derives *a new formalism* from already existing ones by their combination, e.g., DEV&DESS [12] or Hybrid Petri nets [1]. Mapping maps formalisms to supporting one so that just the supporting formalism is interpreted. Wrapping connects simulators of different formalisms so that each model is interpreted by its simulator and simulators communicate with each other by means of a compatible interface. To embed OOPN to DEVS, we have chosen *wrapping*. DEVS is very suitable to serve as a basic component platform for multiparadigmatic simulations because it rigorously defines the component interface.

### 4.1. PNTALK ARCHITECTURE

PNtalk is a simulation framework based on OOPN implemented in Smalltalk environment. Its architecture has been designed as metalevel, where the domain level represents models in OOPN and the metalevel represents elements of the domain level (i.e. OOPN classes and objects ) as metaobjects in an implementation environment. We may distinguish two views on the metalevel in PNtalk—the representation of OOPN elements and the system dynamism. For use in a description of embedding OOPN, only two metaobjects are important—$PNO$ representing an OOPN object and $PNW$ (called world) representing a collection of objects collaborating in the simulation. Both metaobjects offer a

metaprotocol for the simulation control. For the metaobject $PNO$ to run, it must be placed into some world—without the world, it has no dynamism.

### 4.2. PNTALK SIMULATION CONTROL

The world is a metaobject $PNW$ with attributes $(time, object, execs, calendar)$[2] where $time$ is a current model time, $objects$ is a list of objects in the world, $execs$ is a set of objects which can do the next step, and $calendar$ is a list of activating records of time-depending events. The world's simulation loop is outlined by the codes in Figure 5. A single simulation step is specified by the method *step*. During each iteration, a special flag *isRunning* is set to *true*. If there is no executable object, the next scheduled event is got from $calendar$ and activated, together with and $time$ update. Otherwise, the message *step* is sent to all executable objects. At the end of the world cycle, the flag *isRunning* is set to *false*. In a case of a state-dependent (i.e. not scheduled, asynchronous) event inside PNtalk, the world is requested for doing *step* by the *nextStep* message.

```
step
  self isRunning: true.
  execs isEmpty
    ifTrue: [self stepTime]
    ifFalse: [execs do: [:o | o step]].
  self isRunning: false.
  execs isEmpty ifFalse: [self nextStep]
```

```
thisObjectIsExecutable: anObj
  execs at: (anObj id) put: anObj.
  self isRunning ifFalse: [
    self nextStep.
  ].
```

Figure 5: Basic operations of the world simulation loop.

When the $PNO$ metaobject receives the *step* message, it chooses one of its enabled events and performs it. Hereby, the list of enabled events can change and the metaobject must test itself. If there is no event enabled, the object removes itself from $execs$ in $PNW$. If there is at least one event, the object sends *thisObjectIsExecutable:* message to the world. It causes the object is put into $execs$.

### 4.3. ADAPTATION OF THE PNTALK WORLD AS A SMALLDEVS COMPONENT

Embedding OOPN into the SmallDEVS framework is done by wrapping of the PNtalk world to a DEVS componet. We have derived a subclass of world $PNDW$ which conforms its interface to the atomic DEVS. The internal transition $\delta_{int}$ in pricile is similar to the operation *step* with one difference—the decision about the next step is left up tp the parent component (see Figure 6). After each step, the DEVS simulator checks the time of the next step by the message *timeAdvance* ($t(a)$). In the OOPN wrapper, works in three steps: (1) if there is at least one executable object, the advance time is $0$; (2) if there is at least one scheduled event with activating time $t$, the time advance is a difference of $t$ and current model time $time$; (3) otherwise, time advance is $infinity$—the world is passivated.

The operation *nextStep* is in SmallDEVS adapted to *se-message*—a mechanism used e.g. in DEVS&DESS for a notification of the root solver about a state event, and also in real-time simulations where some DEVS components serve as interfaces to the outer world which can generate incidental events. The operation *nextStep* thus serves as a request for internal transition.

[2]This notion of metaobjects is for more obvious presentation of the basic idea – of course, the concrete implementation consists of added attributes.

```
intTransition
  self isRunning: true.
  execs isEmpty
    ifTrue: [self stepTime]
    ifFalse: [execs do: [:o | o step]].
  self isRunning: false.
```

```
timeAdvance
  execs isEmpty
    ifFalse: [^0]
    ifTrue: [
      (t := calendar nextTime) isNil
        ifFalse: [ ^ t - time ]
        ifTrue: [ ^ Float infinity ]
    ].
```

Figure 6: Basic operations of the adapted world simulation loop.

The sets of input and output event values $X$ and $Y$ from atomic DEVS description are specified as structured sets. So we can use named input and output ports for these variables. DEVS components communicate with each other through their ports: when a new object is placed to the output port of a component, it is carried to the appropriate input port of the another component. The way how to relate DEVS ports with OOPN is to map ports and places. Ports are mapped onto places by their names—a port named $n$ is mapped onto a place named $n$. The external transition ($\delta_{ext}$) and the output function ($\lambda$) of DEVS-compatible OOPN wrapper are implemented in $PNDW$ as descibed in Figure 7.

```
extTransition
  ports := self inputPortNames.
  ports do: [:pName || v net |
    v := self peekFrom: pName.
    net := mainObject objectNet.
    (net placeNamed: pName)
      add: v mult: 1.
  ].
  mainObject test.
```

```
outputFnc
  ports := self inputPortNames.
  ports do: [:pName || v place |
    place := mainObject objectNet
      placeNamed: pName.
    (place size > 0)
      ifTrue: [
        v := place itemAt: 1.
        place take: v mult: 1.
        self poke: v to: pName.
      ].
  ].
```

Figure 7: The operations mapping DEVS ports onto OOPN places.

## 5. CONCLUSION

We have outlined the OOPN formalism embedding into DEVS. By the approach presented here, (1) we enrich the formalism of OOPN with an asynchronous communication through asynchronous ports, and (2) we have got a combination of OOPN and DEVS formalism, where the atomic DEVS can be specified by high-level Petri Nets. The DEVS wrapper enriches OOPN by a potential of flexible interoperability based on asynchronous communication and it allows OOPN to be DEVS compatible.

An integration of OOPN and DEVS allows for multiparadigm modeling as well as better application of formal models in the model base design approach, which is a part of our research too. The proposed component-based interoperability of OOPN with other formalisms embedded to DEVS enables us to connect OOPN models with the real world surroundings, needed, e.g., for hardware-in-the-loop simulation. Note that SmallDEVS puts some another requirements on the components, such as serialization, copy-and-paste manipulation protocol and ability to be investigated and edited at runtime. A description of solving these requirements is beyond the scope of this paper.

Our experience indicates that in most of cases only simple OOPN-based components will be used—just one OOPN object having only object net (without methods) specifies an atomic DEVS component. Nevertheless, there are applications requiring a high level of dynamism and concurrency. They need to be specified by non-trivial OOPN object referring other objects which can live inside the PNtalk world wrapped into the DEVS framework. Such applications are, e.g., agent systems. We have already experimented with a simple agent system specified by OOPN. Petri nets have been used there for a representation of segments of plans of the agents as well as for the specification (and implementation) of the overall agent architecture.

**Acknowledgment.** *This work was supported by the Czech Grant Agency under the contracts GP102/07/P306, GA102/07/0322, and Ministry of Education, Youth and Sports under the contract MSM 0021630528.*

## REFERENCES

[1] ALLA H., DAVID R., *Discrete, Continuous, and Hybrid Petri Nets*, Springer 2005.

[2] BARROS F.J., ZEIGLER B.P., FISHWICK, P.A., *Multimodels and Dynamic Structure Models: An Integration of DSDE/DEVS and OOPM*, Proceedings of the 30th conference on Winter simulation, Washington, D.C. 1998, pp. 413-420.

[3] ČEŠKA M., JANOUŠEK V., VOJNAR T., *PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling*, Computer Aided Systems Theory and Technology, Vol. 1333 (1997) of Lecture Notes in Computer Science, pp. 591–610.

[4] CHOW A.C.H., ZEIGLER B.P., *Parallel DEVS: a parallel, hierarchical, modular, modeling formalism*, Proceedings of the 26th conference on Winter simulation, Orlando, Florida 1994, pp. 716-722.

[5] GENRICH H.J., LAUTENBACH K., *System Modelling with High-Level Petri Nets*, Theoretical Computer Science, Vol. 13, 1981.

[6] GOLDBERG A., ROBSON D., *Smalltalk 80: The Language*, Addison-Wesley 1989.

[7] HU X., ZEIGLER B.P., *Model Continuity in the Design of Dynamic Distributed Real-Time Systems*, IEEE Transactions on Systems, Man and Cybernetics, Part A, Vol. 35 (2005), pp. 867–878.

[8] JANOUŠEK V., *PNtalk: Object Orientation in Petri nets*, Proc. of European Simulation Multiconference, Prague, Czech Republic 1995, pp. 196-200.

[9] JANOUŠEK V., KIRONSKÝ E., *Exploratory Modeling with SmallDEVS*, Proc. of ESM 2006, Toulouse, France 2006, pp. 122-126.

[10] JANOUŠEK V., KOČÍ R., *PNtalk: Concurrent Language with MOP*, Proceedings of the CS&P'2003 Workshop, Warsawa 2003, pp. 271-282.

[11] JENSEN K., *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag 1994.

[12] ZEIGLER B., *Embedding DEV&DESS in DEVS: Characteristic Behavior of Hybrid Models*, DEVS Integrative M&S Symposium, Huntsville, Alabama, USA 2006.

[13] ZEIGLER B., KIM T., PRAEHOFER H., *Theory of Modeling and Simulation*, Academic Press (2nd edition) 2000.