Defining and Implementing Dynamic Semantics of Object Oriented High Level Petri Nets

Marius Brezovan

Faculty of Automation Computers and Electronics, University of Craiova
1100 Craiova, Romania

Abstract. This paper deals with the dynamic semantics of a new proposed formalism, called Object Oriented High Level Petri Nets (OOHLPN). In the first part of the paper, the formal definition of OOHLPN is briefly introduced, and an example of using OOHLPN is provided. The second part of the paper presents the dynamic semantics of OOHLPN. First, the semantics of Extended High Level Petri Nets with Objects (EHLPNO) is described. These nets are based on standard of High Level Petri Nets, enriched with some object-oriented concepts. Next, the semantics of OOHLPN is described using notions of state, marking, step and step firing. The third part of the paper presents some algorithms for implementing the dynamic semantics of OOHLPN.

1 Introduction

Petri nets are graphical and mathematical tools used in different areas where the notion of events and concurrence appear. To specify large systems, many high level Petri nets formalisms has been developed, such as Predicate/Transition Nets [9], Colored Petri Nets [10] and Algebraic Nets [12]. During the last years, there have been many proposal of introducing object-oriented features into the frame of Petri nets to increase the power for modeling concurrent and distributed systems: PROT nets [1], POTs and POPs nets [11], OBJSA nets [4], CO-OPN [8], Cooperative Nets [2, 3], LOOPN language [11], Pntalk language [7].

In this paper a new proposed formalism called Object Oriented High Level Petri Nets (OOHLPN) is presented, which is based on the standard of High Level Petri Nets. These nets are constructed using two elements: (a) to allow tokens from a Petri net to be instances of some classes, and (b) to transform a Petri net into a class, allowing its instances to exist inside another Petri nets. To define OOHLPN, first Extended High Level Petri Nets with Objects (EHLPNO) are defined, and also the classes associated to EHLPNO.

The second part of the paper describes the dynamic semantics of OOHLPN in two steps: first the semantics of EHLPNO is presented, and next the semantics of the entire OOHLPN is described. The third part of the paper presents some methods and algorithms for implementing the dynamic semantics of OOHLPN.

2 The Formal Definition of Object Oriented High Level Petri Nets

The definition of Object Oriented High Level Petri Nets is made in two steps: (a) by modifying High Level Petri Nets, allowing tokens to have objects as values [6], and (b) by transforming a High Level Petri Net into a class, allowing Petri nets to exist inside other objects.

In the first step, a class of nets called *Extended High Level Petri Net with Objects* (EHLPNO) is defined, which represent High Level Petri Nets enriched with some object orientation concepts such as: (a) creating new objects inside transitions, when they fire, and (b) calling public methods of objects inside transitions.

Objects can exist inside Petri nets as token assigned to some variables. To create an object inside a transition, a syntactical construction called *object creation specification* is used, which is defined as:

x.create(<params>)

where <params> is a list of expressions of terms compatible with the formal arguments of the method *create* of the class associated to *x*. The set of all object creation specifications is denoted by CREATE.

In terms of Petri nets, a method can be viewed as a subnet of an enclosing net, having an input place containing the input tokens of the method, and an output place for the resulting tokens. Transitions contained in the subnet represent the actions performed by that method. The subnet associated to a method is not necessary to be explicit specified; only its input and output places.

A *method* is defined as a triplet of the form (m, #m, m#), where *m* represents the name of the method, *#m* its associated input place, and *m#* the associated output place. The place *#m* is also an input place of the enclosing net, and *m#* is also an output place of the enclosing net. At most one of the elements #m and m# can be the empty symbol λ, in the case when there aren't input or output tokens for that call. The set of all methods is denoted by METH.

There are considered two types of method call. Let *t* be a transition and *x* a variable associated to *t*, which is bound to an object *ob*. A *method call* is a syntactical construction having one of the following forms:

- x.m(a), which represent an *asynchronous call*,
- b $\leftarrow$ x.m(a), which represent a *synchronous call*,

where:

- *m* is a public method from the class associated to the object *ob*;
- *a* is an expression containing output variables of *t*; when evaluated, it results a multiset of tokens over the input place #m;
- *b* is an expression containing input variables for *t*; when evaluated, it results a multiset of tokens over the output place m#.

In the following, MC will denote the set of all method calls.

The semantics is defined for each type of method call. Let *t* be a transition containing a call of a method *m* of an object bound to a variable *x* associated to an arc incident to *t*.

1. For an asynchronous call the calling transition *t* is extended with an extra output arc connected to the input place of the called method *m*, as is presented in Figure 1.

2. For a synchronous call the calling transition *t* is extended with a subnet containing a start transition *t#start*, an end transition *t#end* and a waiting place *t#wait*. The start transition is connected with the input place of the called method *m*, and the end transition is connected with the output place of *m*, as is presented in Figure 2.

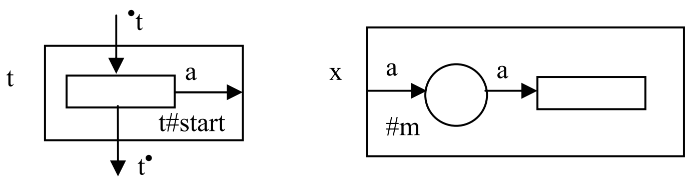

Fig. 1. Asynchronous call

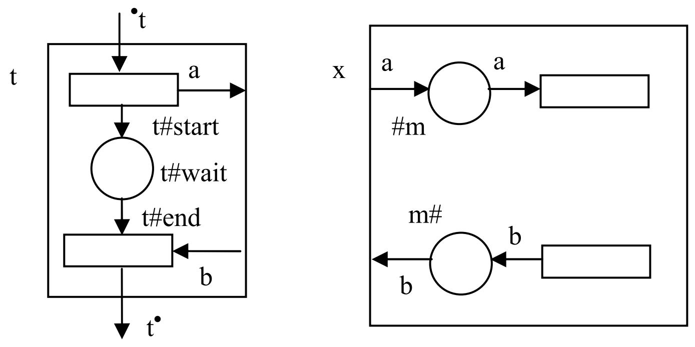

Fig. 2. Synchronous call

Data types contained into a class are managed through algebraical specifications [8]. To allow introducing the notion of inheritance, well-structured hierarchical algebraical specifications [5] will be used instead of algebraical specifications.

Definition 1. Let Sp=(S, O, V, EQ) be a well-structured hierarchical algebraical specification. An *Extended High Level Petri Net with Objects* is a tuple:

$$\text{Ehlpno} = (\text{NG, Sig, V, H, Type, AN, } M_0\text{, ON, Y}),$$

where:

- Hlpn = (NG, Sig, V, H, Type, AN, M_0) is a standard High Level Petri Net;
- ON = (TCreate, TCall) is the object net annotation:
  - TCreate:T$\rightarrow\wp$(CREATE), is a function which assign object creation specifications to transitions;
  - TCall:T$\rightarrow$MC$\cup\{\lambda\}$, is a function assigning at most one method call to each transition;
- *Y* is a set of S-indexed variables, disjoint from V, representing the set of initialization parameters.

When use an instance of an EHLPNO, the variables from the initialization set must be replaced by the actual values.

Let *Ehlpno* be an EHLPNO and σ an assignment for variables from *Y*, $\sigma \in \sigma_Y$. An *instance* of *Ehlpno* is denoted by (Ehlpno, σ) and it is a standard High Level Petri Net obtained from *Ehlpno* by performing the following actions:

- the substitution of all transitions having method calls with their corresponding subnets, according to figures 1 and 2,
- the substitution of all occurrences of variables from *Y* with their bindings.

The set of all instances of *Ehlpno* is denoted by EInst(Ehlpno).

An EHLPNO can be transformed into a class. In the following, SORT will denote the set of all sorts, and $SORT_{Cl}$ will denote the set of the types of all classes, $SORT_{Cl} \subseteq SORT$.

Definition 2. Let *Ehlpno* be an EHLPNO. A *class associated* with *Ehlpno* is a triplet:

$$Cl = (Sp, I, Ehlpno).$$

where:

- Sp = (S, O, V, EQ) is the hierarchical algebraical specification associated with *Ehlpno*;
- I = (s, $\le_I^s$, M) represents the interface of the class, where:
  - s $\subseteq$ $SORT_{Cl}$ represents the sort of the class;
  - $\le_I^s$ is a partial order relation on the set $SORT_{Cl}$, specifying the ascendancies of the current class *s*;
  - M $\subseteq$ METH, represents the public methods of *Ehlpno*;

The set of all classes associated with EHLPNO will be denoted with CLASS.

An instance of a class associated with an EHLPNO is defined in terms of algebraical specifications. For an algebraical specification Sp = (S,O,V,EQ), an *instance* of *Sp* is a pair (Sp, σ), where $\sigma \in \sigma_V$. The *instance set* of the specification *Sp* is denoted by SInst(Sp).

Let Cl = (Sp, I, Ehlpno) be a class associated to Ehlpno = (NG, Sig, V, H, Type, AN, M_0, ON, Y). An *instance* of the class *Cl* is an object *ob* defined as follows:

$$ob = ((Sp,\sigma), I, (Ehlpno,\sigma_1))),$$

where: $\sigma \in \sigma_V$, $\sigma_1 \in \sigma_Y$, $(Sp,\sigma) \in SInst(Sp)$, $(Ehlpno,\sigma_1) \in EInst(Ehlpno)$.

When using objects inside a Petri net, a variable is not bound to an object, but to a reference of the object. This reference can be retained by a map, denoted *ref*, which associates to each object an *object identifier*. In the following OID will denote the set of all object identifiers, and OB will denote the set of all objects. The map *ref* is a bijective function defined as:

$$ref: OID \to OB,$$
$$ref(oid) = ob = ((Sp, \sigma), I, (Ehlpno, \sigma_1)),$$

where $\sigma \in \sigma_V$, $\sigma_1 \in \sigma_Y$.

The notion of inheritance is defined on the set $SORT_{Cl}$ as an order relation.

Let $s_1, s_2 \in SORT_{Cl}$ be two types of classes associated with two EHLPNOs $Ehlpno_1$ and $Ehlpno_2$ and let Cl_1 = (Sp_1, (s_1, $\le_I^{s1}$, M_1), $Ehlpno_1$), Cl_2 = (Sp_2, (s_2, $\le_I^{s2}$, M_2), $Ehlpno_2$) be their associated classes. The class of type s_2 *inherits* the class of type s_1 and it is denoted $Cl_1 \le_I Cl_2$ iff:

- $(s_1, s_2) \in \le_I^{s1}$
- $M_1 \subseteq M_2$

- $Sp_1 \leq_H Sp_2$
- $Ehlpno_1 \approx_{Sem} Ehlpno_2$

The relation $\leq_H$ is defined over well-structured hierarchical algebraical specification. $Sp_1 \leq_H Sp_2$ iff Sp_2 is obtained from Sp_1 by using zero or more operators $\cup$ and $+$. The relation $\approx_{Sem}$ concerns the semantics of the associated nets. $Ehlpno_1 \approx_{Sem} Ehlpno_2$ iff their reachability graphs are (strongly) bisimilar in the sense of Milner and Park.

Let $SP = \{Sp_{1, \ldots,} Sp_n\}$ be a set of well-structured hierarchical algebraical specifications. A *class hierarchy* associated to *SP* is a set of classes together with the inheritance relation:

$$H = \{Cl_k = (Sp_k, I_k, Elpno_k) \mid Sp_k \in SP\}$$

A *root* of the hierarchy *H* is a class $Cl = (Sp, (s, \leq_I^s, M), Ehlpno)$ for which $\leq_s = \Phi$.

Definition 3. Let $SP = \{Sp_{1, \ldots,} Sp_n\}$ be a set of well-structured hierarchical algebraical specifications. An *Object Oriented High Level Petri Net* associated to *SP* is a class hierarchy with a single root:

$$Oohlpn = \{Cl_0, Cl_1, .., Cl_n\},$$

where, for $k=1,\ldots n$, $Cl_k = (Sp_k, I_k, Elpno_k)$, $Sp_k \in SP$, so that Cl_0 is the main root of *Oohlpn*.

The main root of an OOHLPN is important when defining the dynamic semantics of these nets. It represents the higher level of abstraction for a modeled system, and an instance of Cl_0 it is the unique object, which exists at the beginning of the dynamic system evolution.

In the following it is presented an example of using OOHLPN for modeling a flexible manufacturing cell. The manufacturing cell has six components, as presented in Figure 3:
- an input buffer IN, which receives the pieces to be manufactured, and an output buffer OUT where the manufactured pieces are stored;
- two manufacturing machines, M_1 and M_2;
- an robot R, used to load and unload the machines;
- an internal buffer B, used to store the unfinished pieces.

The pieces are supposed to require two ordered operations, op_1 and op_2, each operation on a single machine: op_1 on machine M_1 and op_2 on M_2.

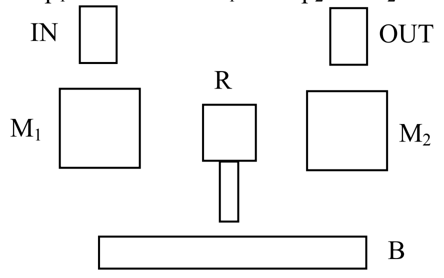

Fig. 3. A simple manufacturing cell

To describe the functionality of the manufacturing cell, the following classes are used:
- the class *Robot*, which describes the functionality of the robot R;

– the class *Machine*, which describes the functionality of the machines M_1 and M_2;
– the class *Cell*, which describes the functionality of the entire manufacturing cell.
The OOHLPN for the manufacturing cell is:

Oohlpn = {Cell, Machine, Robot}

where *Cell* is the main root class of the hierarchy, each class being hierarchical independent.

The EHLPN of the class *Robot* is presented in the Figure 4. It has a single public method, (move, #move, move#), which is used when the robot must move a piece from a source place to a destination place. All places have a single structured type for tokens with three components, (x, y, z), representing 3D coordinates, and the place *InitPos* is the only place with a non-empty marking.

The EHLPN of the class *Machine* is presented in the Figure 5. It has a single public method, (oper, #oper, oper#), which is used when an operation on the machine has to be performed. For uniformity, all places have structured tokens as in the class *Robot*, and the place *Available* is the only place with a non-empty marking.

To describe the EHLPNO associated with the class *Cell*, a rule-based like language is used. Each transition is described by a rule, where the antecedent and the consequent represent some places of the net. The following six rules describes the functionality of the manufacturing cell (the first action of each rule consequent represents a method call):

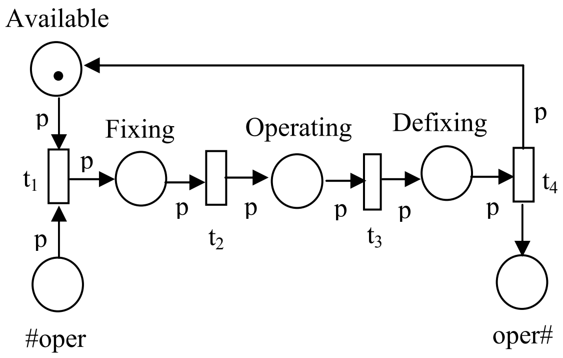

Fig. 4. The Extended High Level Petri Net associated with the class Machine

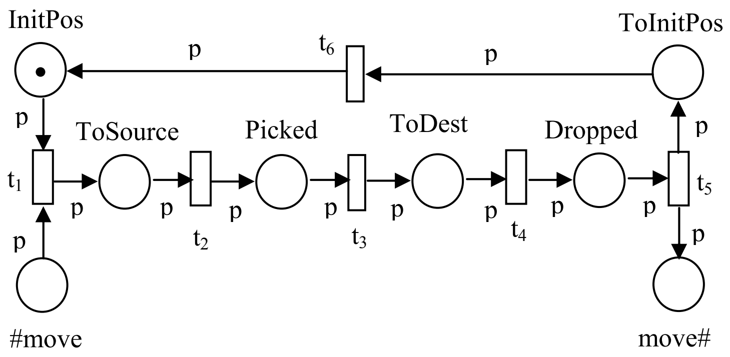

Fig. 5. The Extended High Level Petri Net associated with the class Robot

t_1: *if* (piece available on IN) $\wedge$ (R available) $\wedge$ (M_1 available) *then*
  (R : move In$\rightarrow M_1$) $\wedge$ (R available) $\wedge$ (M_1 loaded) *end*
t_2: *if* (M_1 loaded) *then* (M_1 : operation) $\wedge$ (piece available on M_1) *end*
t_3: *if* (piece available on M_1) $\wedge$ (R available) *then*
  (R : move $M_1\rightarrow$B) $\wedge$ (R available) $\wedge$ (piece available on B) *end*
t_4: *if* (piece available on B) $\wedge$ (R available) $\wedge$ (M_2 available) *then*
  (R : move B$\rightarrow M_2$) $\wedge$ (R available) $\wedge$ (M_2 loaded) *end*
t_5: *if* (M_2 loaded) *then* (M_2 : operation) $\wedge$ (piece available on M_2) *end*
t_6: *if* (piece available on M_2) $\wedge$ (R available) *then*
  (R : move $M_2\rightarrow$OUT) $\wedge$ (R available) $\wedge$ (piece available on OUT) *end*

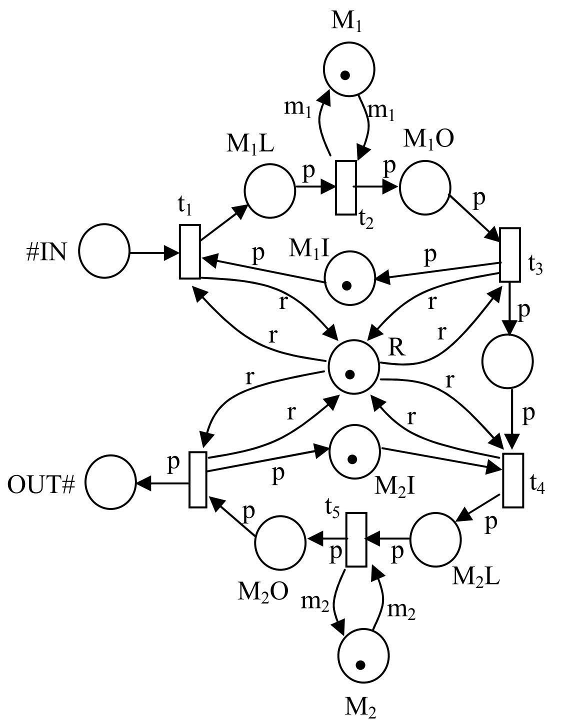

Fig. 6. The Extended High Level Petri Net associated with the class Robot

The EHLPNO associated with *Cell* is presented in Figure 6. The place R contains instances of the class *Robot* as tokens, while the places M_1 and M_2 contain instances of the class *Machine*. The places #IN, OUT#, B, M_1I, M_1L, M_1O, M_2I, M_2L, M_2O have structured tokens with eight components, each component being a 3D point corresponding to the eight places where the robot can move (the input an the output place of the machine M_1, the input an the output place of the machine M_2, the input an the output place of the buffer B, the input place of the buffer IN and the output place of the buffer OUT) denoted by {m_1i, m_1o, m_2i, m_2o, bi, bo, in, out}. The places with non-empty initial marking are M_1, M_2, R, M_1 and M_2I.

The method calls associated with the transitions are the followings:

t_1: $p.m_1i \leftarrow r.move(p.in)$	t_4: $p.m_2i \leftarrow r.move(p.bo)$
t_2: $p.m_1o \leftarrow m_1.oper(p.m_1i)$	t_5: $p.m_2o \leftarrow m_2.oper(p.m_2i)$
t_3: $p.bi \leftarrow r.move(p.m_1o)$	t_6: $p.out \leftarrow r.move(p.m_2o)$

The class *Cell* has two public methods, (IN, #IN, λ) and (OUT, λ, OUT#), used to put a piece into the cell, or to extract a finished piece from the cell.

3 Defining Dynamic Semantics of Object Oriented High Level Petri Nets

An OOHLPN describes a class of systems with similar properties. To model a particular system, some instances of the classes contained into the OOHLPN hierarchy have to be created, and next their concurrent activities have to be handled.

Let $S \subseteq SORT_{Cl}$ be a set of class types. The set of all object identifiers for the classes associated to *S*,

$$Cl(S) = \{((S, O, V, EQ), I, Ehlpno)|\ s \in S\},$$

can be structured like a Sig-algebra, called *identifiers algebra,* over which a partial order relation induced by the order relation $\leq_I$ is overlapped. For *S*, the partial order relation of classes is determined as follows:

$$\leq_I^S = \cup_{s \in S}(\leq_I^s).$$

Now, the informal construction of the family of sets $(OID_s)_{s \in S}$ is the following:

$$OID_s = \{oid(s, k) \mid k= 1, 2, ... \},$$

where *k* is the order number of the current object. So, the first created object from the class $(Sp, (s, \leq_I^s, M), Ehlpno) \in Cl(S)$ class will have the reference (s,0), the second (s,1), … etc.

In the following, the dynamic semantics of OOHLPN will be defined in two steps: (a) for an instance of a class associated to an EHLPNO, and (b) for the entire set of instances representing an OOHLPN.

For handling the dynamic evolution of the objects from an OOHLPN, the notion of dynamic link will is used. Let $Oohlpn=\{Cl_1,\ldots,Cl_n\}$ be an OOHLPN. A *dynamic link* is a pair (oid, ref(oid)), for which there exists a class Cl = ((S,O,V,EQ), I, Ehlpn)$\in$Oohlpn, and a sort $s \in S$, such that: $oid \in OID_s$.

To create an object *ob* of a class, an object identifier *oid* have to be created, and a reference to that object, ref(oid) = ob, so that (oid, ref(oid)) represents a dynamic link for the object *ob*.

The current state of an object is done by the current marking of its associated EHLPNO.

Definition 4. Let *Ehlpno* be an EHLPN, Cl=(Sp, I, Ehlpn) be a class associated with *Ehlpno*, and ob=((Sp,σ), I, (Ehlpno,σ_1))) be an object of *Cl* with a dynamic link (oid, ref(oid)). The *current state* of *ob* is a pair (oid, M), where *M* is the current marking of the instance (Ehlpno,σ_1)$\in$EInst(Ehlpno).

The notion of occurrence mode of a transition can be defined. The occurrence mode specifies the bindings for the variables associated to a transition. Denoting by *t* a transition, the following notations will be used:

VarIn(t), representing the set of variables attached to input arcs of *t*,
VarOut(t), representing the set of variables attached to output arcs of *t*,
$Var(t) = VarIn(t) \cup VarOut(t)$, representing the set of all variables attached to *t*.

Definition 5. Let *Ehlpno* be an EHLPN, $Ei = (Ehlpno, \sigma_1) \in EInst(Ehlpno)$ one of its instances and *t* be a transition of *Ei*. An assignment α of the set of variables Var(t) is called *occurrence mode* for *t* if there exists an evaluation Val_α in $TERM(O \cup V)$ so that the following statements hold:

- if $TCreate(t) \neq \Phi$, then $\forall x \in VarOut(t)\text{-}VarIn(t)$, $\alpha(x)=x.create(a_1,...,a_n)$, so that ref(oid)=ob, where $x.create(a_1,...,a_n) \in TCreate(t)$;
- the selector TCond(t) can be evaluated;
- all multisets associated to incident arcs of *t* can be evaluated.

To define the enabling condition for a transition, the multiset of terms associated with an arc can be extended by using a map denoted *arc*:

$arc: (P \times T) \cup (T \times P) \to TERM(O \cup V) \cup \Phi$,
$arc(u, v)=A(u, v)$, if $(u, v) \in F$,
$arc(u, v)=\Phi$, if $(u, v) \notin F$

A transition *t* is enabled in a marking *M* for a particular assignment for its variables if there are enough tokens into the input places of the transition.

Definition 6. Let $ob=((Sp,\sigma), I, (Ehlpno,\sigma_1)))$ be an object with a dynamic link (oid,ref(oid)), (oid,M) be its current state, *t* be a transition in the instance $(Ehlpno,\sigma_1)$ and α be an occurrence mode of *t*. The transition *t* is *enabled* with the occurrence α in the state (oid,M) and is denoted $(oid,M)[t{:}\alpha\rangle$, iff the following condition is satisfied:

$\forall\ p \in P,\ Val_\alpha(arc(p, t)) \leq M(p)$

An enabled transition *t* may fire.

Definition 7. Let $ob=((Sp,\sigma), I, (Ehlpno,\sigma_1)))$ be an object with (oid, ref(oid)) a dynamic link, (oid, M) be its current state, and *t* an enabled transition in the instance $(Ehlpno, \sigma_1)$. The *firing* of *t* leads *ob* to the next state (oid, M^1), which is denoted by $(oid, M)[t{:}\alpha\rangle(oid, M^1)$, and defined as follows:

$$\forall p \in P,\ M^1(p) = M(p) - Val_\alpha(arc(p, t)) + Val_\alpha(arc(t, p))$$

Remark. When a transition *t* contained into an object *ob* fires, this cause the changing of the current state of *ob*, and also the changing of the state of other objects having public methods, if these methods are called inside of *t*:

- If *t* is a start transition and *#m* is the input place of the called method *m*, contained into an object:
$$ob_1 = ref(oid_1) = ((Sp_1, \sigma), I_1, (Ehlpno_1, \sigma_1),$$
having a dynamic link $(oid_1, ref(oid_1)$, and the state (oid_1, M_1), then its new following state will be:
$$(oid_1, M_1)[t{:}\alpha\rangle(oid_1, M_1{}^1)$$
where:

$$M_1^{\ 1}(\#m) = M_1(\#m) + Val_\alpha(arc(t, \#m))$$

- If *t* is an end transition and *m#* is the output place of the called method *m*, contained into an object:
$$ob_1 = ref(oid_1) = ((Sp_1, \sigma), I_1, (Ehlpno_1, \sigma_1),$$
having a dynamic link $(oid_1, ref(oid_1))$, and the state (oid_1, M_1), then its new following state will be $(oid_1, M_1)[t{:}\alpha\rangle(oid_1, M_1^{\ 1})$, where:
$$M_1^{\ 1}(\#m) = M_1(\#m) - Val_\alpha(arc(m\#, t))$$

Now, the dynamic semantics of an entire OOHLPN can be described. The state of a modeled system is represented by the states of all current objects specified by their dynamic links. At the beginning of the system evolution, there is a unique object, which is instance of the root of the OOHLPN.

Definition 8. Let *Oohlpn* be an OOHLPN.
(1) The *initial state* of *Oohlpn* is the unique dynamic link associated to the root of *Oohlpn*:
$$SA_0 = \{(oid_0, ref(oid_0)) \mid oid_0 \in OID_0\}$$
(2) A *state* of *Oohlpn* is a finite set of dynamic links:
$$SA = \{(oid, ref(oid)) \mid \exists Cl=((S,O,V,EQ),I,Ehlpn) \in Oohlpn, \exists s \in S : oid \in OID_s\}$$

Each object from the current state of an OOHLPN has an associated EHLPNO with a current marking. All these markings represent the current marking of the OOHLPN.

Definition 9. Let *Oohlpn* be an OOHLPN.
(1) A *marking* of *Oohlpn* corresponding to a state *SA* of *Oohlpn* is the set of all object states:
$$OM = \{(oid, M) \mid (oid, ref(oid)) \in SA\}$$
(2) The *initial marking* of *Oohlpn* in the initial state SA_0 is the state associated to the object oid_0:
$$OM_0 = \{(oid_0, M_0) \mid (oid_0, ref(oid0)) \in SA_0\}$$

The marking of an OOHLPN is changing every time when the state of an object from the current state of the net is changed.

Definition 10. Let *SA* be a state of an OOHLPN *Oohlpn*, and *OM* be its marking in the state *SA*.
(1) A *step Y* is a set of enabled transitions contained into objects of the state *SA*, each object from *SA* having exactly one enabled transition in *Y*.
(2) The step *Y fires* and leads *Oohlpn* to a new state SA_1 and a new marking OM_1 under this state, iff all transition from *Y* fire. The new state is :
$$SA_1 = SA - Destroy(Y) \cup Create(Y),$$
where: Destroy(Y) represents the set of all objects destroyed in the current state *SA*, and Create(Y) represents the set of all objects created in the current state *SA*:
$$Destroy(Y) = \cup_{t\in Y}\{(\sigma(x), ref(\sigma(x))) \mid x \in VarIn(t) - VarOut(t)\}$$
$$Create(Y) = \cup_{t\in Y}\{(\sigma(x), ref(\sigma(x))) \mid x.create(a_1, ..., a_n) \in C(t)\};$$
The new marking is:
$$OM_1 = \{(oid_1, M_1) \mid (oid_1, ref(oid_1)) \in SA_1\}$$
where:

– if $(oid_1, ref(oid_1)) \in SA$ and $(oid_1, M) \in M$, then $\exists t \in Y$ with occurrence α and $(oid_1,M)[t:\alpha\rangle(oid_1, M_1)$,
– if $(oid_1, ref(oid_1)) \notin SA$, then: $(oid_1, M_1) = (oid_1, M_0)$.

Passing to a new state with a new marking is denoted: $SA[Y\rangle SA_1$, $OM[Y\rangle OM_1$.

4 Implementing Dynamic Semantics of Object Oriented High Level Petri Nets

The implementation of the dynamic semantics of OOHLPN is made in two steps: in the first step the semantics of EHLPNO is implemented and next the semantics of the entire OOHLPN.

An optimized version of token-player algorithm is used for implementation, which is based on *launch places* [13]: each transition has an associated launch place, chosen from the set of its input places. If this place has an empty marking then doubtlessly the transition isn't enabled, else the transition is possible to be enabled.

An Extended High Level Petri Nets with Objects *Ehlpno* represents the body of a class *Cl*. From a programming language view, all classes associated with EHLPNO are derived from a distinct base class, called *CPNO*. The algorithms which implement the dynamic semantics of EHLPNO are member functions of *CPNO*.

The constructor of *CPNO* expands in the initial net each transition having method calls with its associated subnets, perform the initial marking of the net, constructs the list *p-enabled* of possible enabled transitions and next it call the *TokenPlayer* procedure which describes the dynamic evolution of the net.

```
procedure CPNO::Create(Y)
  NetExpanding()
  InitMarking(Y)
  InitEnabled()
  TokenPlayer()
end
```

The *TokenPlayer* algorithm can be described as follows:

```
procedure CPNO::TokenPlayer()
  enabled-list ← GetEnabledTransitions()
  while enabled-list ≠ Φ do
    t ← SelectEnabledTransition(enabled-list)
    FiringTransition(t)
    enabled-list ← GetEnabledTransitions()
  od
end
```

The function *SelectEnabledTransition* chose an enabled transition from the set of all current enabled transitions of the net, and it represents non-determinism in the dynamic evolution of the net. The function *GetEnabledTransitions* determines all enabled transitions, based on the function *Enabled*:

```
logical function CPNO::Enabled(t)
  for *∀p∈°t do
    if ¬(Match(Eval(A(p,t)),M(p)) then
      return false
    fi
  od
  if Type(t)=end-call then
    tok ← GetObject(TCall(t)).
         OutputMethodPlace(GetMethod(TCall(t)))
    if ¬Match(Eval(OutputVariables(TCall(t))),tok) then
      return false
    fi
  fi
  if ¬Eval(TC(t)) then
    return false
  fi
  return true
end
```

The function *Eval* evaluates a multiset of variables according to their bindings, and the function *Match* performs a matching action between a multiset of variables and a multiset of constants, and, if possible, performs the binding operation for variables.

For a public method (m, #m, m#) defined in the interface *I* of the class *Cl*, the functions *OutputMethodPlace* and *InputMethodPlace* return the current marking of the places *m#* and *#m* respectively.

The procedure *FiringTransition* performs the firing operation and updates the list *p-enabled* under the new marking.

```
procedure CPNO::FiringTransition(t)
  for *∀x∈VarIn(t)-VarOut(t) do
    DeleteOject(x)
  od
  for *∀p∈°t do
    DelTokens(M(p), Eval(A(p,t)))
    DelEnabled(p)
  od
  if Type(t)=end-call then
    tok ← GetObject(TCall(t)).
            OutputMethodPlace(GetMethod(TCall(t)))
    GetObject(TCall(t)).
            DelTo-
kens(tok,Eval(OutputVariables(TCall(t)))
  fi
  for *∀c∈TCreate(t) do
    CreateOject(Variable(c),Params(c))
  od
  for *∀p∈t° do
```

```
      AddTokens(M(p), Eval(A(t,p)))
      AddEnabled(p)
    od
    if Type(t)=start-call then
      tok ← GetObject(TCall(t)).
              InputMethodPlace(GetMethod(TCall(t)))
      GetObject(TCall(t)).
              AddTo-
  kens(tok,Eval(InputVariables(TCall(t)))
    fi
  end
```

The functions *AddEnabled* and *DelEnabled* update the list *p-enabled*. The function *CreateObject* calls the constructor for a specified object and run indirectly its *TokenPlayer* method.

An OOHLPN *Oohlpn* is a class hierarchy with a single root, and the modeling of a concurrent system is made by the set of current dynamic links, representing the current state of *Oohlpn*, and by the marking of the current state:

$\text{Oohlpn} = \{\text{Cl}_1, ..., \text{Cl}_n\}$,

$\text{SA} = \{(\text{oid}, \text{ref(oid)}) \mid \exists \text{Cl}=(\text{S,O,V,EQ})\in\text{Oohlpn}, \exists s\in \text{S}: \text{oid}\in\text{OID}_s\}$

$\text{OM} = \{(\text{oid}, \text{M}) \mid (\text{oid}, \text{ref(oid)})\in\text{SA}\}$

During the dynamic evolution of the modeled system, the objects contained into the current state of the OOHLPN have a relatively independent evolution, the only interactions between these objects being method calls.

To implement the dynamic evolution of an OOHLPN, the notions of state, marking, step and step firing have to be managed.

To manage the current state *SA* of an OOHLPN, means to manage the concurrent evolution of the objects contained into *SA*. That is a platform dependent problem, and there are three main solutions:

(a) For a sequential programming environment, a virtual concurrency can be realized using a small kernel program that maintains a list of active objects and provides primitives to add/remove objects to/from the list. The kernel must have a scheduler, which activate each object by calling its *TokenPlayer* function.

(b) For an environment which supports threads, all objects are running concurrently on the same machine, each object in its own thread.

(c) For a distributed environment, all objects are running in parallel, eventually on different machines.

Initially there is a single object corresponding to the dynamic link (oid_0, $\text{ref}(\text{oid}_0)$), which represents the initial state SA_0. The initial marking of the net associated to oid_0, and its dynamic evolution will create new objects and will destroy others.

Except for the object which is the instance of the root class of the hierarchy, all other objects are created by the function *CreateObject*, which is called within the functions *InitMatking* and *FiringTransitions* of an already created object. The objects can be destroyed only by the function *DeleteObject*, called within the function *FiringTransition* of an object.

When use threads, the procedure *TokenPlayer* of a new created object *ob*$_k$ runs in the thread of the other object *ob*$_j$ which creates *ob*$_k$. The procedure *TokenPlayer* of the object *ob*$_0$ runs in the main thread.

A step is a set of enabled transitions contained into objects of the current state, one enabled transition in each object. Because the function *SelectEnabledTransition* selects only one enabled transition from the net associated to the current object, it results that the set of transitions returned by the function *SelectEnabledTransition* of all current alive objects represents the current step. The *firing* of the current step is made by the call of the function *FirirngTransition* of all current alive objects.

In conclusion, to implement the dynamic semantics of an OOHLPN it is necessary:

– to implement the dynamic semantics of EHLPNO;
– to coordinate a system of concurrent objects, which represent instances of classes associated to EHLPNO.

5 Conclusions

In this paper, specification formalism for discrete event systems, called Object Oriented Petri High Level Nets is proposed. The OOHLPN formalism has been proposed for the need to encapsulate the object-oriented methodology into the Petri net formalism. In addition, like Predicate/Transition nets, OOHLPN preserve the similarity with the rule-based systems. For this reason, a simple rule-based language used to describe OOHLPN can be developed, each rule of the rule base being expressed as a transition in the correspondent OOHLPN.

There are described algorithms for implementing the semantics oh EHLPNO, and methods to implement the semantics of OOHLPNO. The manner in which the dynamic semantics of OOHLPN have been defined allows a simple way to construct a OOHLPN simulator.

The C++ language can be used for OOHLPN implementation. All the OOHLPN concepts can be implemented using data structures and methods of the standard C++ language, so that the resulted class hierarchy can be portable on different C++ environments. Any class from an OOHLPN hierarchy can be associated to a C++ class derived from the CPNO class. The OOHLPN translator will generate one C++ class by generating the constructor to that class, so that the simulator for an OOHLPN will be generated in a semi-compiled way.

Future work will be carried out in three main directions:

– to study thoroughly the connection between rule-based systems and Object Oriented High Level Petri Nets and to develop a method of implementing dynamic semantics of OOHLPN using an inference engine for rule-based systems;
– to determine a High Level Petri Net which is semantic equivalent to an OOHLPN;
– to develop a graphical and text-based language for OOHLPN and to build a tool supporting this language.

References

1. Baldassari M., Bruno G.: An Environment for Object-Oriented Conceptual Programming Based on PROT Nets, Springer LNCS, Vol.340 (1988) 1-19.
2. Bastide R.: Cooperative Objects: a Formalism for Modelling Concurrent Systems, Ph.D. Thesis, University of Toulouse III, (1992)
3. Bastide R., Sibertin-Blanc C, Palanque P.: Cooperative Objects: a Concurrent, Petri-Net Based, Object-Oriented Language, *Proceedings IEEE International Conference on Systems, Man and Cybernetic* (1993) 286-292.
4. Battiston E., DeCindio F., Mauri G.: OBJSA Nets: a Class of High Level Nets having Objects as Domain, Springer LNCS, Vol.340 (1988) 20-43.
5. Biberstein O., Buchs D.: An Object-Oriented Specification Language based on Hierarchical Algebraic Petri Nets, In: Wieringa R., Feenstra R. (eds.): Working papers of the *International Workshop on Information System Correctness and Reusability* (1994) 47-62.
6. Brezovan M.: A Formal Definition of High Level Petri Nets with Objects, *Annals of the University of Craiova* 26(2):45-54 (2002)
7. Ceska M., Janusek V.: A Formal Model for Object Oriented Petri Nets Modeling, Advances in System Science and Application (1997)
8. Ehrig H., Mahr B.: Fundamentals of Algebraic Specifications, *EATCS Monograph in Theoretical Computer Science*, Vol. 6, Springer (1985)
9. Genrich H.J.: Predicate/Transition Nets, Springer LNCS, Vol.254 (1987) 207-247

11. Jensen K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, *EATCS Monographs on Theoretical Computer Science*, Vol.1: Basic Concepts (1992)
12. Lakos C.A.: Object Petri Nets, Definition and Relationship to Colored Nets, University of Tasmania, Technical Report TR94-3 (1994)
13. Reisig W.: Petri Nets and Algebraic Specifications, *Theoretical Computer Science*, 80:1-34 (1991)
14. Valette R.: Petri nets for control and monitoring: specification, verification, implementation, Proceedings Workshop “Analysis and Design of Event-Driven Operations in Process Systems”, London (1995)