ENCAPSULATED MULTI-METHODS AND MULTIPLE DISPATCHING IN OBJECT ORIENTED HIGH LEVEL PETRI NETS

Marius Brezovan *

* *University of Craiova, Software Engineering Department*

Abstract: In recent years, several proposals tried to associate object-oriented formalisms and Petri nets into a single framework which combine the expressive power of both approaches. This paper presents a Petri net formalism called *Object Oriented High Level Petri Nets* (OOHLPN), and the implementation of encapsulated multi-methods and multiple dispatching in this formalism. The encapsulated multi-methods and multiple dispatching are important feature of OOHLPNs, in addition to separating the inheritance and subtyping hierarchies.

Keywords: Petri nets, object-oriented Petri nets, object-oriented languages, formal methods

1. INTRODUCTION

In order to specify large systems, many high level Petri nets formalisms has been developed, such as Predicate/Transition Nets (Genrich, 1987), Colored Petri Nets (Jensen, 1992) and Algebraic Nets (Reisig, 1991). 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 modelling concurrent and distributed systems: PROT nets (Baldassari and Bruno, 1988), OBJSA nets (Battiston *et al.*, 1988), CO-OPN (Biberstein *et al.*, 2001), Cooperative Nets (Bastide *et al.*, 1993), LOOPN language (Lakos, 1994).

This paper presents a new formalism called *Object Oriented High Level Petri Nets* (OOHLPN) , which is based on the standard of High Level Petri Nets and object-oriented notions and methodologies.

In the first part of paper, the main notions concerning the OOHLPN formalism are presented. The second part of the paper describes the implementation of two important notions in this formalism: the encapsulated multi-methods, and multiple dispatching mechanism.

2. OBJECT ORIENTED HIGH LEVEL PETRI NETS

Because because subtyping inheritance are orthogonal mechanisms, concerned with the manipulation of types and implementations, respectively (Cook *et al.*, 1990; Taivalsaari, 1996), one of main feature of the OOHLPN formalism is the separation of the class notion into two distinct elements:

- An *interface*, which is used to define an object type;
- An *implementation module*, which is used to specify an object type implementation.

Due to this separation, Object Oriented High Level Petri Nets are defined in three different stages:

- definition of object types and interfaces;
- definition of implementation modules,
- definition of the entire collection of object types and modules, which represent an OOHLPN.

Throughout this paper, we assume a universe U, which includes several disjoints sets:

$$U = SORT \cup OID \cup METH$$

where $SORT$ is the set of all sorts, OID is the set of all identifiers associated to objects, and $METH$ is the set of public method names of object types. The set of sorts, $SORT$, is partitioned in three disjoint sets, $SORT_D$, $SORT_{Ob}$ and $SORT_{Ref}$, corresponding to the names of non object-oriented data types, the names of object types and the names of reference types respectively.

In order to specify the methods of objects, the set $METH$ is considered as a sorted set:

$$(METH_{\hat{c}w,s})_{\hat{c}\in SORT_{Ref},w\in SORT^*,s\in SORT\cup\{\epsilon\}}$$

For each method name $m \in METH_{\hat{c}w,s}$, $\hat{c}$ represents the reference type to the object containing the method, w is the sequence of sorts representing the input values, and s is the sort of the returned value.

An *encapsulated multi-method* is an overloaded method associated with an object type, which contains several methods (branches) having the same name. Let c be an object type. An *encapsulated multi-method* of c with n branches is a pair (m, t), where:

(a) $m \in \bigcap_{i=1}^{n} METH_{\hat{c}w_i,s_i}$ is the name of the multi-method, with $w_1, \ldots, w_n \in SORT^*$ and $s_1, \ldots, s_n \in SORT \cup \{\epsilon\}$;
(b) $t = \{\hat{c}w_1 \rightarrow s_1, \ldots, \hat{c}w_n \rightarrow s_n\}$ represents its type.

where $w \rightarrow s$, with $w = s_1 \ldots s_n$, represents a function type, having input values of sorts $s_1 \ldots s_n$ and output value of sort s. The set of all possible encapsulated multi-methods of c is denoted by $MMETH(c)$

Definition 2.1. An *interface*, $Intf$, defining an object type c is a tuple

$$Intf = (c, \leq^S, Mt, Create)$$

where:

- c is the name of the interface;
- $\leq^S \subseteq \{(c, s) \mid s \in SORT_{Ob}\}$ is a partial order specifying the subtype relation associated to c;
- $Mt \subseteq MMETH(c)$ is a finite set of encapsulated multi-methods of c.
- $Create = (create, t_{create})$ is the set *create* methods of c.

The set of all interfaces is denoted by $INTF$.

An *implementation* provides the realization of an object type behavior, and it defines a set (or class) of objects having the same internal structure and behavior. The implementation module of an object type is realized in the OOHLPN formalism by using a class of Petri nets called *Extended High-Level Petri Net with Objects* (EHLPNO), which represent high level Petri nets enriched with some object orientation concepts, such as creating new objects inside transitions when they fire, and calling public methods of objects inside transitions:

- A *send action* is a syntactical construction having the following form:

  $$x.m(a_1, \ldots, a_n)$$

  where m is the name of a method of the object ob, $a_1, \ldots, a_n \in TERM(O \cup V)$ are expressions containing input variables of t, and b is an output variable of t.
- A *retrieve action* is a syntactical construction having the following form:

  $$b \leftarrow x.m$$

- An *instruction for creating objects* can be used:

  $$\langle variable \rangle = new\ \langle impl \rangle (\langle params \rangle)$$

  where $\langle variable \rangle \in VarOut(t) - VarIn(t)$ represents a variable associated to the newly created object, $\langle params \rangle$ is a list of expressions containing input variables of t, and $impl$ is the name of the implementation module of the created object.
- An *assignment action* for t is a syntactical construction having the following form:

  $$v \leftarrow e$$

  where $v \in VarOut(t)$ is a variable having a sort $s \in SORT_D$, and e is an expression with the same sort, $e \in TERM(O \cup V)_s$.

A general assignment action of the form

$$b \leftarrow x.m(a_1, \ldots, a_n)$$

must be divided in two actions associated with two distinct transitions: a first transition containing a send action, an intermediate place waiting for the result, and a second transition containing a retrieve action.

Definition 2.2. Let SG be a set of order-sorted signatures, $Sig \in SG$ a Boolean order-sorted signature, $Sig = (S, \leq, O)$, $H = (S_H, \leq, O_H)$ an order-sorted Sig-algebra, IT a set of well-defined interfaces, $Intf \in IT$ an interface, $Intf = (c, \leq^S, Mt, Create)$, defining the object type c. An *implementation module* of the interface $Intf$ is a triple:

$$Impl = (sm, Ehlpno, Intf, Inh)$$

where:

(i) $sm \in SORT_{Imp}$ is the name of the module;

(ii) *Ehlpno* is an Extended High-Level Petri Net with Objects;
(iii) $Inh \in SORT_{Imp} \cup \{undef\}$ specify implementation inheritance relation of $Impl$.

To simplify notations, several functions will be used:

- $ImplSort(Impl)$ will denote the name of $Impl$,
- $Ehlpno(Impl)$ will denote the EHLPNO associated to $Impl$,
- $Interface(Impl)$ will denote the interface that $Impl$ implements,
- $Inherit(Impl)$ will denote the inherited implementation module of $Impl$.

The definition of OOHLPN uses the notions of subtype and inheritance hierarchy.

Definition 2.3. Let IT be a set of well-defined interfaces, IM a set of well-defined modules that implement interfaces from IT. An *object-oriented system* associated to IT and IM is a triple:

$$OS = (IT, IM, Inst)$$

where $Inst : IM \rightarrow \wp(OID)$ is a function which associates a set of object identifiers to each implementation module, such that if $Impl_i, Impl_j \in IM$, $Impl_i \neq Impl_j$, then $Inst(Impl_i) \cap Inst(Impl_j) = \emptyset$.

The object identifier sets $Inst(Impl_i), i = 1, \ldots, n$ are disjoint sets in the case of unrelated implementation, in order to prevent two instances of implementations to have the same object identifier.

Definition 2.4. Let $OS = (IT, IM, Inst)$ be a object-oriented system as in the above definition. An *Object Oriented High Level Petri Net* associated to OS is a triple:

$$Oohlpn = (OS, Impl_0, oid_0)$$

where $Impl_0 \in IM$ is a root of the inheritance hierarchy of the object-oriented system, called the initial implementation module of $Oohlpn$, and $oid_0 \in Inst(Impl_0)$ is the object identifier associated to the initial object of $Oohlpn$.

The initial implementation module $Impl_0$ of an OOHLPN represents the higher level of abstraction for a modelled system, and its initial object is the unique instance of $Impl_0$, which exists at the beginning of the dynamic system evolution.

3. ENCAPSULATED MULTI-METHODS AND MULTIPLE DISPATCHING

In the OOHLPN formalism, each instance of an object type has an associated object identifier which can be used in different net annotations.

Let $Oohlpn = (OS, Impl_0, oid_0)$ be an OOHLPN, associated to an object-oriented system, $OS = (IT, IM, Inst)$. The sets of object types and implementation module names associated to OS are denoted by $ObjType^{OS}$ and $ImplSort^{OS}$ respectively. Let c be an object type of OS, such that $ObjType^{-1}(c) \in IT$. The set of all object identifiers of c is denoted by Oid_c and it is defined as:

$$Oid_c = \bigcup_{Impl \in Interface^{-1}(ObjType^{-1}(c))} Inst(Impl)$$

The set of all object identifiers of OS is denoted by Oid^{OS} and defined as:

$$Oid^{OS} = \bigcup_{c \in ObjType^{OS}} Oid_c$$

Because the sets of object identifiers $Inst(Impl)$ from an object-oriented system are disjoint, one can define a function, denoted by $CurrentImpl$, which returns for each object identifier its associated implementation module. The restriction of $CurrentImpl$ to the set Oid^{OS} is bijective:

$$CurrentImpl : Oid^{OS} \rightarrow IM,$$
$$CurrentImpl(oid) = Impl,\ oid \in Inst(Impl)$$

By using the function $CurrentImpl$, the object type and the implementation module name associated to an object identifier can be determined.

Using the above notation, the set of all reference values associated to a reference type $\hat{c}$ is defined as the set:

$$Ref_c = Oid_c$$

An action $Ac(t) \in SEND \cup RET$ allows objects to communicate. In order to allow multiple dispatching, OOHLPN uses a run-time dispatching mechanism, implemented also with Extended High Level Petri Nets with Objects.

For an action $Ac(t) \in SEND \cup RET$, the implementation module of the receiver object cannot be known at compile-time. Because at compile-time the name of the called method, and the name of the object type are both known, the dispatcher mechanism uses a dispatcher module for each object type from the object type hierarchy of an OOHLPN.

Let $Intf = (c, \leq^S, Mt, Create)$ be an interface associated to the object type c, having k encapsulated multi-methods:

$$Mt = \{(m_1, t_1), \ldots, (m_k, t_k)\}$$

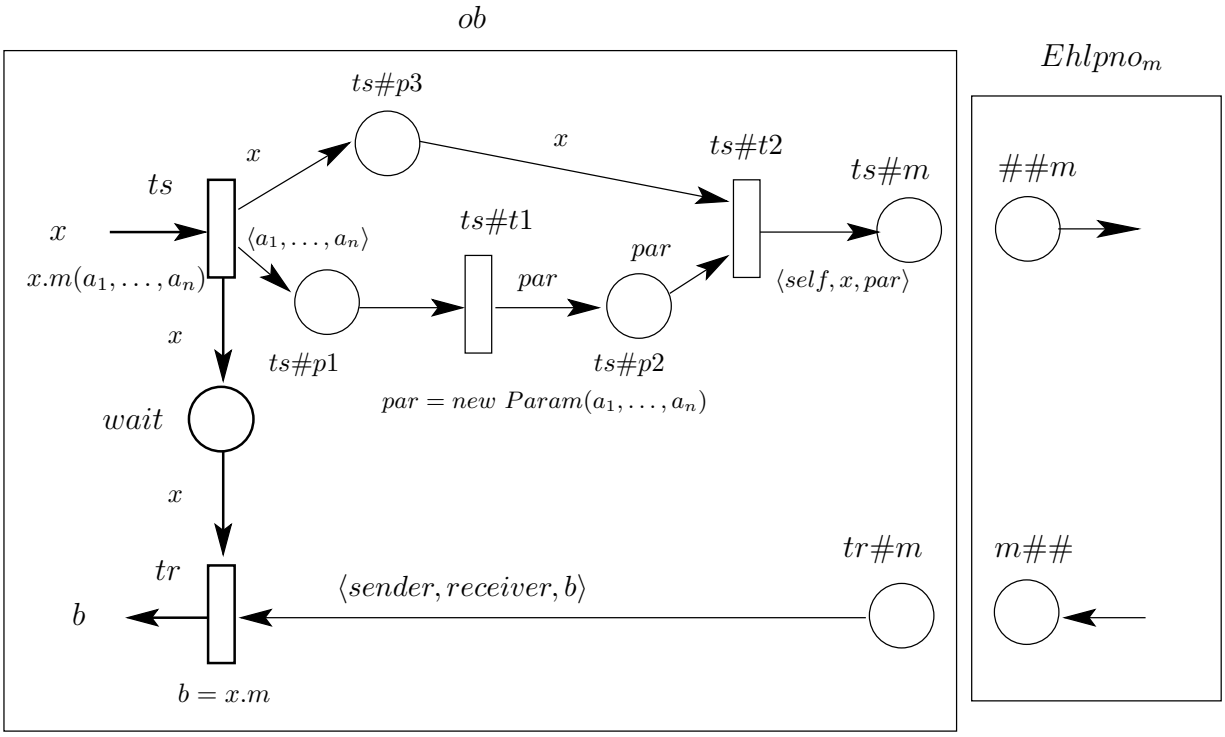

Fig. 1. A method call

The dispatcher module associated to $Intf$ is denoted by $Disp_{Intf}$ and it is a collection of distinct Extended High Level Petri Nets with Objects,

$$Disp_{Intf} = \{Ehlpno_{m_1}, \ldots, Ehlpno_{m_k}\}$$

where each net $Ehlpno_{m_i}$ is associated to a multi-method m_i from Mt.

A method call involving two actions of the form $x.m(a_1, \ldots, a_n)$ and $b = x.m$, where x is a reference to the receiving object, is performed by using the extended Petri net, $Ehlpno_m$, associated to the multi-method m from the dispatching module corresponding to the object type of x. Figure 1 specify the semantics of a such method call from an object ob, containing a sending transition denoted by ts, a waiting place denoted by $wait$, and receiving transition denoted by tr. The dispatching sub-module, $Ehlpno_m$, has an input place, denoted by $\#\#m$, and an output place denoted by $m\#\#$.

The Petri net corresponding to the implementation module associated to the object ob is extended with the following elements:

- four places, $ts\#p1$, $ts\#p2$, $ts\#p3$ and $ts\#m$, two transition, $ts\#t1$, $ts\#t2$, and seven corresponding arcs, used to send the input information for the multi-method m;
- one place, $tr\#m$, and a corresponding arc, used to receive the output information from the multi-method m.

The transition $ts\#t1$ is used to create an object of the type $Param$, which is a list of parameters containing the arguments $a_1, \ldots, a_n$ of the called method. The object type $Param$ is used because different branches of a multi-method, m, can have different number and types of arguments, and its unique input place, $\#m$, requires an unitary treatment of these arguments. The transition $ts\#t2$ is used to create the compound value $\langle self, x, par \rangle$ which is sent to the dispatching module. The places $ts\#m$ and $\#\#m$ are fused, and the current Petri net containing the transition ts is in this way connected to the Petri net $Ehlpno_m$ associated to the multi-method m.

The place $tr\#m$ is used to receive from the dispatching sub-module $Ehlpno_m$ the result of the called method and the two additional references, to the caller object and to the called object, respectively. The places $tr\#p$ and $m\#\#$ are also fused.

Let $Intf = (c, \leq^S, Mt, Create)$ be an interface, and $(m, t) \in Mt$ one of its encapsulated multi-methods. The EHLPNO of the dispatching sub-module associated to the multi-method (m, t) is presented in Figure 2. Its input and output places are denoted by $\#\#m$ and $m\#\#$ respectively.

The multiple dispatching mechanism performs two actions:

- a traditional method lookup as in single dispatching languages, in order to determine (in presence of subtype polymorphism) the correct object whose method is called;

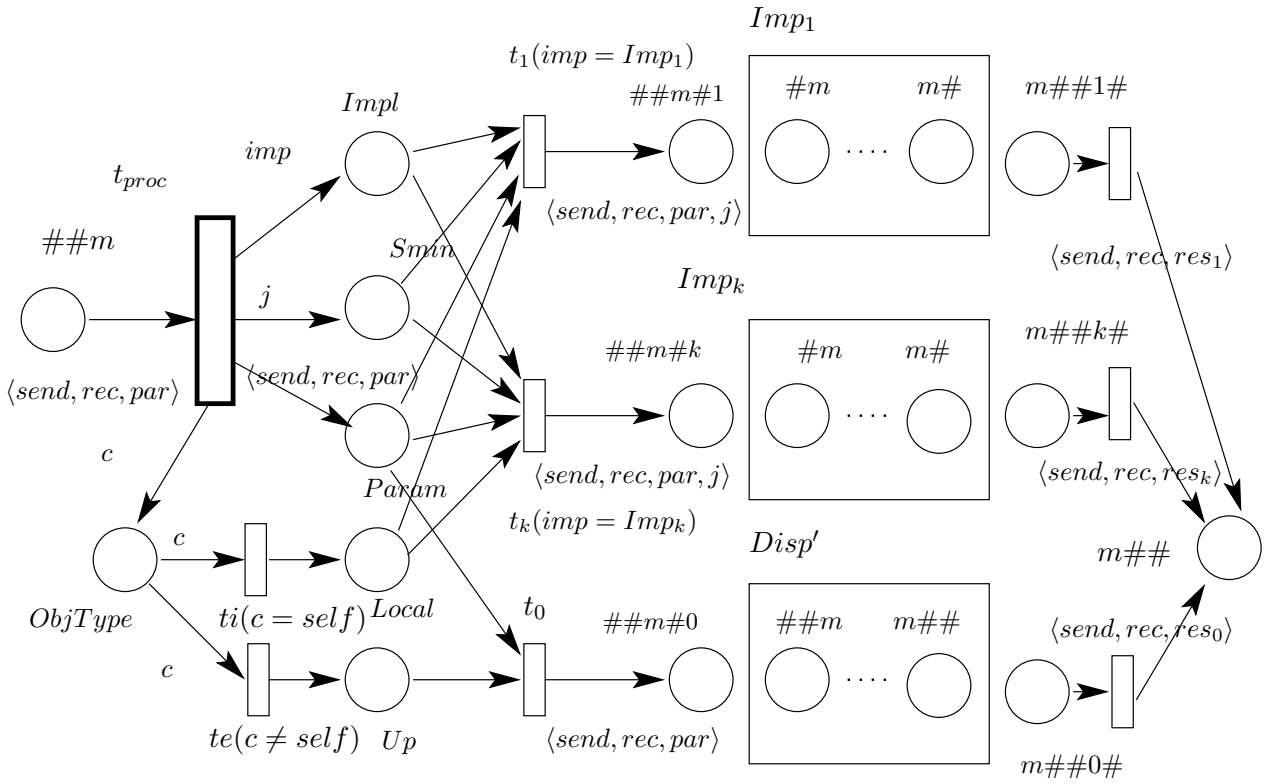

Fig. 2. A dispatching sub-module associated to a multi-method

- a branch selection algorithm of multiple dispatching languages, based on the run-time types of message arguments.

Let $t = \{\hat{c}w_1 \rightarrow s_1, \ldots, \hat{c}w_n \rightarrow s_n\}$ be the type of the encapsulated multi-method (m, t), and $\hat{c}w \rightarrow s$ the function type of the actual arguments. The best matching branch j is selected such that (Bruce *et al.*, 1996):

$$w_j = min_{1 \leq i \leq n}\{w_i \mid w \leq w_i\} \qquad (1)$$

The dispatching sub-module presented in Figure 2 implements these two actions as follows:

(a) The transition denoted by t_{proc}, that represents in fact a sub-net (it is denoted in figure as a transition for simplicity), determines from the input information $\langle send, rec, par\rangle$ the following three elements:
  · the appropriate object type of the receiver object (variable c), extracted from the reference of the receiver object (variable rec); this action is necessary in presence of subtype polymorphism;
  · the name of the appropriate implementation module (variable imp) containing the called method, extracted also from the reference of the receiver object;
  · the best matching branch (variable j) determined as in Equation (1), extracted from the variable par and from the type of the encapsulated multi-method.

(b) The place $ObjType$ and transitions ti and te allow to determine if the current object type of the receiver object is equal to the object type of the interface containing the multi-method.

(c) In the case when the object type of the receiver object is the same as the object type of the interface containing the multi-method, the transitions t_1, ..., t_k allow to send to all implementation modules (denoted by Imp_1, ..., Imp_k) the input information of the dispatching sub-module (variables $send$, rec and par), and in addition, the best matching branch (variable j). The the best matching branch is used in each implementation module in order to determine the appropriate method, as presented in Figure 3. The pairs of places $(\#\#m\#1, \#m)$, ..., $(\#\#m\#k, \#m)$, and $(m\#, m\#\#1\#)$, ..., $(m\#, m\#\#k\#)$ are then fused.

(d) In the case when the object type of the receiver object is different than the object type of the current interface, the transition t_0 allows to resend the input information to the dispatching sub-modules (associated to the same multi-method) of the parent interfaces. In Figure 2 is represented for simplicity a single dispatching sub-module, denoted by $Disp'$. The pairs of places $(\#\#m\#0, \#\#m)$ and $(m\#\#, m\#\#0\#)$ are also fused.

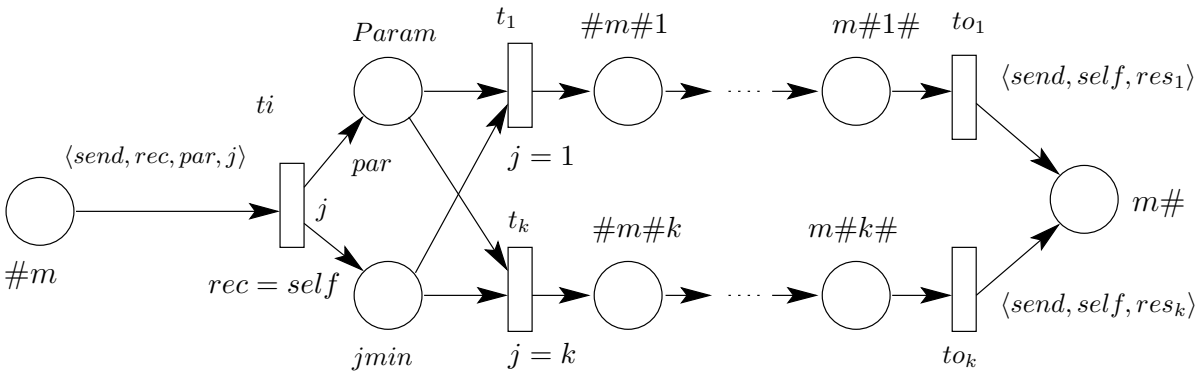

Fig. 3. Implementation of an encapsulated multi-method

(e) The results from all implementation modules and all dispatching sub-modules are collected and stored in the output place $m\#\#$.

The structure of an implementation of a encapsulated multi-method with k branches is presented in Figure 3, and it contains the following elements:

- the k subnets associated to the k branches; the subnet corresponding to the i^{th} branch is represented between places $\#m\#i$ and $m\#i\#$;
- a dispatching mechanism, represented by transitions ti, t_1, ..., t_k, in order to select the best matching branch;
- a collecting mechanism, represented by transitions to_1, ..., to_k, which allow to collect the results from each branch.

4. CONCLUSION

In this paper, we presented a new class of Petri nets, called Object Oriented High Level Petri Nets and the implementation of their encapsulated multi-methods and multiple dispatching. The OOHLPN formalism has been proposed for the need to encapsulate the object-oriented methodology into the Petri net formalism.

Comparing with other object-oriented Petri net formalisms, OOHLPNs have two important features:

- The OOHLPN formalism allows two distinct hierarchies, for subtyping and for inheritance, because inheritance and subtyping are distinct notions;
- The OOHLPN formalism uses encapsulated multi-methods instead of ordinary methods, and it has also a multiple dispatching mechanism.

OOHLPNs have also a third important feature, a garbage-collector mechanism, but its implementation is not presented in this paper.

REFERENCES

Baldassari, M. and G. Bruno (1988). An environment for object-oriented conceptual programming based on prot nets. *LNCS* **340**, 1–19.

Bastide, R., C. Sibertin-Blanc and P. Palanque (1993). Cooperative objects: A concurrent, petri-net based, object-oriented language. In: *Proc. of the IEEE International Conference on Systems, Man and Cybernetics*. pp. 286–292.

Battiston, E., F. DeCindio and G. Mauri (1988). Objsa nets: A class of high level nets having objects as domain. *Lecture Notes in Computer Science* **340**, 20–43.

Biberstein, O., D. Buchs and N. Guelfi (2001). Object-oriented nets with algebraic specifications: The co-opn/2 formalism. *Lecture Notes in Computer Science* **2001**, 70–127.

Bruce, K., L. Cardelli, G. Castagna, The Hopkins Objects Group, G. Leavens and B. Pierce (1996). On binary methods. *Theory and Practice of Object Systems* **l(3)**, 221–242.

Cook, W., W. Hill and P. Canning (1990). Inheritance is not subtyping. In: *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*. pp. 125–135.

Genrich, H.J. (1987). Predicate/transition nets. *Lecture Notes in Computer Science* **254**, 207–247.

Jensen, K. (1992). Coloured petri nets. basic concepts, analysis methods and practical use. *EATCS Monographs on Theoretical Computer Science*.

Lakos, C. A. (1994). Object petri nets, definition and relationship to colored nets. Technical Report TR94-3. University of Tasmania.

Reisig, W. (1991). Petri nets and algebraic specifications. *Theoretical Computer Science* **80**, 1–34.

Taivalsaari, A. (1996). On the notion of inheritance. *ACM Computing Surveys* **28(3)**, 438–479.