Object-oriented modelling with high-level modular Petri nets

Bui Thanh, Cécile and Klaudel, Hanna

Object-oriented modelling with high-level modular Petri nets

Cécile Bùi Thanh and Hanna Klauedl

LACL, Université Paris 12
61, avenue du général de Gaulle
94010 Créteil, France
{bui,klauedl}@univ-paris12.fr

Abstract. In this paper, we address the problem of expressing object-oriented concepts in terms of Petri nets. This is interesting, first, as a possibility of representing concurrent system specifications written in object-oriented formalisms or languages with Petri nets, and second, as a way allowing automated verification of the obtained Petri net using existing reachability analysis tools.

We start from an existing parallel specification language having a modular Petri net semantics and we extend it with object-oriented features inspired from Java and C++. The translation of these new extensions into the Petri net domain is given using a class of modular coloured Petri nets and includes, in particular, a treatment of inheritance and of dynamic binding.

Keywords. Object-orientation, coloured Petri nets, semantics.

1 Introduction

In this paper, we propose a way to express object-oriented concepts, like inheritance and dynamic bindings, in terms of Petri nets. The motivation is to provide a translation of concurrent system specifications written in object-oriented formalisms or languages into Petri nets, and thus, to allow automated verification of the obtained Petri net using existing reachability analysis tools [7, 18, 15].

The starting point of our approach is the high-level parallel specification language B(PN)$^2$ [1, 9] (Basic Petri Net Programming Notation). It comprises most traditional concepts of parallel programming, like parallel composition, iteration, guarded commands, procedures and communications. Thanks to its simplicity, it can easily be used as a basis for various extensions and the results found for it may then be applied to “real-life” languages. Another advantage of B(PN)$^2$ is that it has already a concurrent formal semantics in terms of a class of high-level (coloured) Petri nets, called M-nets [2]. The particularity of M-nets is that they are provided with a set of composition operations and allow to represent large (possibly infinite) systems in a compact and structured way. Moreover, B(PN)$^2$ and M-nets are implemented in the PEP toolkit [7], allowing to simulate a modelled system and also to verify its properties via model checking.
In this paper we propose an extension of B(PN)$^2$, called the Basic Object-Oriented Notation (BOON), having a syntax inspired from Java [8] and C++ [17], and a semantics in terms of M-nets. This extension allows for defining classes with their own fields (attributes and methods), single class inheritance, polymorphism, and dynamic binding. The proposed semantics is modular, in particular, each class is represented by a module, itself composed of various submodules, each one representing either an attribute or a method of the class, or a mechanism intended for handling inheritance or management of instances (objects). All these modules are combined thanks to the powerful M-net synchronisation mechanism leading to a (large but structured) coloured Petri net. It may be seen as an alternative to other Petri net based formalisms capable to express object-oriented concepts, which often use more complex net classes. This is the case, for instance, for Object Petri Nets (OPN) [10], whose nets are enriched with net tokens, or for CO-OPN [3] and CLOWN [5], which use algebraic Petri nets (nets extended with algebraic data types).

The paper provides also a discussion concerning the soundness of this extension, in particular, concerning the correctness of the handling of dynamic bindings. In this respect, it perceptibly improves the first attempts in defining an object-oriented version of B(PN)$^2$ proposed in [13, 12].

2 Syntax and Semantics of B(PN)$^2$

B(PN)$^2$ is a parallel programming language comprising shared memory parallelism, channel communication and allowing the nesting of parallel operators, blocks and procedures. The following is a fragment of the syntax of B(PN)$^2$ (with keywords typeset in bold face, non-terminals in roman face and italic denoting values supplied by the program):

\[
\text{program ::= program block} \\
\text{block ::= begin scope end} \\
\text{scope ::= com | decl | scope} \\
\text{com ::= (expr | P(arglist) | block | (com | do alt-set od} \\
\text{| com || com | com ; com}
\]

An atomic command is a B(PN)$^2$ expression "$\langle\text{expr}\rangle$", i.e., a term constructed over operators, constants (from a given set $\mathcal{V}$) and program variables, which can be executed if the expression evaluates to true. A program variable$^1$ $v$ can appear in an expression as $'v$ (pre-value) or $v'$ (post-value), denoting respectively its value just before and just after performing the command during the program execution. It may also appear just as $v$ if the command does not change its value (if it is just read). Thus, for example, $'(v > 0 \land v' = w)$ corresponds to an atomic statement which requires the variable $v$ to be greater than zero in which case the value of the variable $w$ is assigned to $v$.

$^1$ Originally, B(PN)$^2$ supports also channel variables, which are omitted here because they may be treated in a different manner in an object-oriented environment.
A command \textit{“com”} is either an atomic command, a procedure call, one of a number of command compositions, or a block comprising some declarations for a command. Parentheses allow to combine arbitrarily the various command compositions.

The domain of relevance of a variable or a procedure identifier is limited to the part of a B(PN)$^2$ program, called \textit{“scope”}, which follows its declaration. As usual, a declaration, in a new block, with an already used identifier results in the masking of the existing identifier by the new one. A declaration of a program variable \( v \) is made with \texttt{“var \( v : V \)”}, where \( V \subseteq \mathbb{V} \) is a set of values, while that of a procedure \( P \) is made with \texttt{“proc \( P \)(parlist)”}, where \texttt{“parlist”} is the list of formal parameters of \( P \).

Besides traditional control flow constructs, like sequence and parallel composition, there is a command \texttt{“do \ldots od”} which allows to express all types of loops and conditional statements (which will not be detailed here).

### 2.1 Existing M-net Based Semantics of B(PN)$^2$

M-nets [2] form a class of high-level (coloured) Petri nets provided with a set of operations giving to them an algebraic structure. Like other high-level Petri net models, M-nets carry the usual annotations on places (sets of allowed tokens), arcs (multi sets$^2$ of annotations) and transitions (guard$^3$). In addition, places have a status (entry, exit or internal) used for net compositions; transitions carry labels used for inter-process communications, which are similar to CCS ones [16] but extended to (multi)sets of actions with arbitrary arity. The communications can be enforced using the operation of \textit{scoping} w.r.t. a set of actions, which intuitively corresponds to a set of binary \textit{synchronisations} involving matching$^4$ pairs of actions, e.g., \texttt{act(x, y)} and \texttt{act(u, 5)}, followed by the restrictions. For instance, the synchronisation w.r.t. \{\texttt{act}\} applied to a net containing the transitions with labels \{\texttt{act(x, y), term(y)}\} and \{\texttt{act(u, 5)}\} will produce in the net a new transition labelled \{\texttt{term(5)}\}, obtained by gluing the two former transitions together, while the restriction w.r.t. \{\texttt{act}\} will remove from the resulting net the transitions whose labels involve \texttt{act} or \texttt{act}. The same operations but w.r.t. \{\texttt{act, term}\} applied to a net containing the transitions with labels \{\texttt{act(x, y), term(y)}\}, \{\texttt{act(u, 5)}\} and \{\texttt{term(z)}\} will produce in this net a new transition (with empty label), corresponding to a three-way synchronisation, and remove from the net all transitions whose labels involve \texttt{act} or \texttt{term}.

The marking of an M-net associates to each place a multiset of values (tokens) from the type of the place and the transition rule is like for other high-level nets. A transition \( t \) can be executed if the inscriptions of its input arcs evaluate to

---

$^2$ A multiset is formally a function \( m \) which gives to each element of a set \( E \) the number of its occurrences. We will use for multisets the extended set notation, \( e.g., \{a, a, b\} \) for \( m(a) = 2, m(b) = 1 \) and \( m(x) = 0 \) for all \( x \in E \setminus \{a, b\} \).

$^3$ A guard is a Boolean expression which plays the role of an occurrence condition.

$^4$ Actions are “matching” if their parameters can be componentwise unified, \( e.g., \) actions \texttt{act(x, 6)} and \texttt{act(u, 5)} are not matching and cannot synchronise.
values which are present in the input places of \( t \) and if the guard of \( t \) evaluates to true. The execution of \( t \) transforms the marking by removing values (accordingly to the evaluation of arc inscriptions) from the input places of \( t \) and by depositing values in its output places.

The M-net semantics of a B(PN)\(^2\) program is defined compositionally in [1] through the semantical function \( \text{Mnet} \). The main idea in describing a block is (i) to juxtapose the nets for its local resource declarations (variables and procedures) with the net for its command followed by a termination net for the declared variables and procedures, (ii) to synchronize all matching data/command transitions and to restrict these transitions in order to make local variables invisible outside the block and (iii) to add the initial marking to the obtained net (typically a black token \( \bullet \) in each initial place).

Each variable or procedure declaration is translated into a corresponding resource M-net. For instance, the declaration of a variable \( v \) of the type \( V \subseteq V \) gives rise to the M-net \( N_{v,V} \) represented in Fig. 1. The current value of the variable \( v \) is stored in the central place of type \( V \) and may be updated using the \{access\_v(x,y)\}-labelled transition. The action access\_v(x,y) describes the change of value of \( v \) from its current value \( x \) to the new value \( y \).

The declaration procedure \( P \) is translated into a procedure resource M-net \( N_P \), composed itself of resource M-nets for all local variables, of the M-net representing the body (command) of the procedure and of the M-net managing various procedures instances. A new procedure instance is started in \( N_P \) thanks to the action call\_P(pid, \( f_1, \ldots, f_n \)), where \( pid \) denotes the procedure instance identifier provided by \( N_P \) and \( f_1, \ldots, f_n \) represents the list of input variables corresponding to the formal parameters intended to be substituted with the effective ones. A call to \( P \) is translated into a call M-net \( N_{\text{Call},P} \), which triggers a new instance of the procedure through the action call\_P(pid, \( e_1, \ldots, e_n \)), where \( e_1, \ldots, e_n \) is the list of values or net variables corresponding to the effective arguments of \( P \). The synchronisation between call\_P(pid, \( f_1, \ldots, f_n \)) and call\_P(pid, \( e_1, \ldots, e_n \)) substitutes the formal parameters with effective ones, which ensures a correct initialisation of each procedure instance. Knowing this will be enough for our purpose; more details concerning the semantics of procedures can be found in [9,6].

Fig. 1. The resource M-net \( N_{v,V} \) of the variable \( v \) of the type \( V \) and the M-net \( N_{\text{access\_v}(x,y),\text{access\_w}(u,w))}(x>y\land y=u) \) of the atomic command \('v > 0 \land v' = w')

Sequential and parallel compositions are directly translated into the corresponding net operations, e.g., \( \text{Mnet}(\text{com}_1;\text{com}_2) = \text{Mnet}(\text{com}_1)\text{Mnet}(\text{com}_2) \), while the semantics of the “do \ldots od” construct involves the M-net iteration operator (not explained here). The semantics of an atomic command “(expr)” is
the M-net $N_{v\gamma}$ where $\alpha$ is a set of actions corresponding to the access to the program variables involved in “expr”, and $\gamma$ is the guard obtained from “expr” with program variables appropriately replaced by net variables, like e.g., $x$ for $'v$, $y$ for $v'$ and $u$ for $w$ in:

$$\text{Mnet}(\langle v > 0 \land v' = w \rangle) = N_{\{\text{access}_v(x,y),\text{access}_w(u,u)\} \mid x > 0 \land y = u}.$$

The M-net above has one transition as shown in figure 1. Its label is used for a communication with the resource nets for variables $v$ and $w$: $x$ is read and $y$ is written with the action $\text{access}_v(x,y)$ (because $v$ is updated) and $u$ is read and written with the action $\text{access}_w(u,u)$ (because $w$ is unchanged). The guard ensures that $x > 0$ and that $y$ is set to $u$.

For instance, the essential part\(^5\) of the M-net semantics of

```
program begin var v, w : V (\langle v' = 3 \rangle(\langle w' = 0 \rangle) ; \langle v > 0 \land v' = w \rangle) end
```

is the initially marked (with one token • in each entry place) M-net

$$(N_{v,V} \mid N_{w,V} \mid (N_{\{\text{access}_v(x,y)\} \mid y = 3} \mid N_{\{\text{access}_w(u,u)\} \mid x > 0 \land y = u})) \subseteq \{\text{access}_v,\text{access}_w\},$$

where $\subseteq \{\text{access}_v,\text{access}_w\}$ is the scoping w.r.t $\{\text{access}_v,\text{access}_w\}$.

3 Object-oriented extension

In order to introduce our object-oriented extension, we fix first a syntax at the B(PN)\(^2\) level. Next, we provide this syntax with a high-level Petri net semantics using M-net algebra. The new notation, inspired from Java and C++, will be called Basic Object-Oriented Notation (BOON).

3.1 Syntax of the object-oriented features

The main extension we propose concerns the introduction in B(PN)\(^2\) of concepts of classes, objects, inheritance, polymorphism and dynamic binding.

A class is a high-level abstraction defined by a set of characteristics and services, called fields. The characteristic fields of a class are called the attributes, and the service ones are called the methods. We assume that each class has a name in a set $C \subseteq V$ and will use the letters $C, D, \ldots$ for denoting classes. A class $D$ may inherit from another class $C$, which means that $D$ has all the fields of $C$, but may overload (redene) some of them and may also have additional fields. In that case, we call $C$ the superclass of $D$, and $D$ a subclass of $C$. When $D$ overloads a field, the new declaration hides the overloaded field for $D$ and its subclasses.

An object is an instance of a class which has its own identity and state. An

\(^5\) The complete semantics takes into account the initialisation and the termination of the variables, which are omitted here.
object of the class $C$ contains all fields defining $C$; we will also say that it is of type $C$. Its state is given by the value of its attributes and may be modified by applying on it one of the methods defined for $C$. It can be created by calling a particular method defined in $C$, called a constructor, and destructed by calling another particular method defined in $C$, called a destructor.

A BOON program is a block containing a list of class definitions and a main command using them. So, a class declaration may be either of the form 
\[ \text{class } C \{ \text{attdecl methoddecl } \}\], where “attdecl” and “methoddecl” are the lists of attribute and method declarations or, if $D$ inherits from $C$, it may be of the form 
\[ \text{class } D : C \{ \text{attdecl methoddecl } \}\].

The attributes may be of two kinds: the standard ones (whose types are subsets of $\mathbb{V}$) or object ones (whose types are classes defined in the program). The corresponding declarations are of the form “\texttt{att a : V}” for a standard attribute of type $V \subseteq \mathbb{V}$ and “\texttt{oatt a : C}” for an object attribute of type $C$. The methods are a kind of procedures declared with the clause “\texttt{method m(parlist) block}”, where “parlist” represents a (possibly empty) list of formal parameters.

The declaration of a subclass $D$ of a class $C$ contains only the declarations of overloaded and additional fields. We assume that a class has a unique\footnote{In fact, it is technically possible to consider user-defined constructors, but for the sake of simplicity and readability, we will consider here only this default constructor.} default constructor (resp. destructor), which just initialises all attributes (resp. deletes) at the creation of the object (resp. at the destruction). Thus, the constructor of a class has as many parameters as the class has attributes.

A standard variable declaration is as before while an object variable is introduced by the clause “\texttt{ovar e : C}”, where $e$ refers to an object of type $C$. Like a standard variable, an object variable $e$ may appear in an expression as ‘$e$, $e$’ or $e$ with analogous meaning. We allow inclusion polymorphism, which means that an object variable $e$ declared of type $C$ may also refer (at a stage of the execution) to an object of any subclass of $C$. An attribute $a$ of an object variable $e$ may appear in an expression as $c(e_{\text{super}})^k.a$, where $k \geq 0$, with possibly a prime before or after to denote its pre- or post-value. If $k \neq 0$, then $c(e_{\text{super}})^k.a$ refers to the attribute $a$ of the $k^{th}$ parent of the class of $e$. If $k = 0$, the expression $c(e_{\text{super}})^k.a$ becomes $c.a$; since the actual type $\kappa$ of the object assigned to $e$ may be a subclass of $C$, $c.a$ is bound to the attribute $a$ declared in the closest class to $D$ in its inheritance tree, which is determined dynamically at the execution time. The latter mechanism is called dynamic binding and exists also for method calling.

We consider new operations at the expression level, namely “\texttt{newc}”, corresponding to the creation of an object of a class $C \in \mathbb{C}$, and “\texttt{del}” corresponding to an object destruction\footnote{Like in C++, it is allowed but not advisable to call explicitly the destructor.}. Typically, they may be used in atomic commands of the form “\texttt{( c = newc(initial) )}”, where “initial” represents the list of initial values for the attributes of the class $C$, or “\texttt{( del(obj) )}”, where “obj” is an object variable or attribute. Moreover, we consider a new command “\texttt{c(e_{\text{super}})^k.m(arglist)}”, which represents the call of the method $m$ on the object $e$ (with the same mean-
ing of \((\text{super})^k\) as for the attributes).

Also, the keyword \texttt{this} can be used in the body of a method of a class \(C\) for referring to the object to which the method is applied. So, in the body of a method \(m\) of class \(C\), an attribute \(a\) may appear as \texttt{this.(super)}\(^k\).\(a\) (and analogously for a method \(m'\) of \(C\)), which has the same meaning as if \texttt{this} was an object variable. The keyword \texttt{this} may also be used in an operation at the expression level. For instance, if a method of a class \(C\) contains the command \((d' = \text{new}_D(\text{this}))\), where \(d\) is an object variable of type \(D\) having an attribute of type \(C\), then if this method is applied to an object \(c\), the execution of the command will initialise the object attribute of \(d\) with \(c\).

4 M-net semantics of object-oriented extensions

Intuitively, the M-net semantics of an object-oriented program involves three parts: the class declarations, class instances management and the main command. All these parts (M-nets) are put in parallel and scoped w.r.t. all communication actions, as sketched below.

\[
\begin{array}{|c|c|c|}
\hline
\text{Declaration of classes} & \text{Management of instances} & \text{Main command} \\
\hline
\end{array}
\]

The part for the class declarations is composed of an M-net for each class declaration and one \textit{inheri\(\text{tance directory}\) M-net. The part devoted to the management of class instances is represented by the \textit{class instances} M-net. As before, the part for the main command is represented by the corresponding M-net.

Except M-nets corresponding to the commands, all M-nets considered from now on are composed of three parts: an entry, an internal and an exit one similarly to the resource M-nets, see Fig. 1. For the sake of readability, we omit the entry and exit parts, which are intended for an adequate initialisation and termination of the internal part. Also, we will omit in the transition labels the empty guards \(\{}\) and the set brackets if the set contains only one element. Moreover, we will signal with an additional arrow \(\uparrow\)\) the action parameters which are intended to be sent (exported), while the remaining parameters are intended to be received, like \textit{e.g.}, in \texttt{new}(\textit{id}, \kappa).

4.1 Management of instances

Each instance of a class is uniquely identified by an identifier \(\textit{id} \in \mathcal{I}\), which can be considered as a pointer, and has also a \textit{type} \(\kappa\), which is the class name.

The object identifiers together with their associated types are managed by the class instances M-net \(N_C\), see Fig. 2, which provides a free identifier from the set \(\mathcal{I}\) to each new object, keeps its actual type, and gets back each identifier released by a destruction. The size of the set of identifiers \(\mathcal{I}\) can be interpreted as the memory size.

We allow the inclusion polymorphism which means that the actual type of an
object variable or attribute may be different from its declaration. For instance, if \( D \) is a subclass of \( C \), the type of \( c \) after execution of the command in the scope “\( \text{new } c : C \ \langle c' = \text{new}_D(\text{initlist}) \rangle' \)” is actually \( D \).

It may also happen that an object variable or attribute is not yet initialised or corresponds to a destructed object. In such a case, this variable or attribute refers to a special (idle) object identified by \( \text{null} \not\in \mathbb{I} \). The identifier \( \text{null} \) is never given to an actual object. However, a destruction command applied to a variable or an attribute identified by \( \text{null} \) is allowed and transparent for the program behaviour.

At the beginning of the execution of the class instances \( \text{M-net } N_i \), the place on the left is marked with a token of each value of its type \( \mathbb{I} \cup \{ \text{null} \} \). The tokens in \( \mathbb{I} \) represent the identifiers available for the creations of objects through the action \( \text{new}(id, \kappa) \), while \( \text{null} \) is used for handling the destruction of idle object variables or attributes through the action \( \text{del}(\text{null}) \). The transition \( t_n \) represents the creation of a new object identified by \( id \in \mathbb{I} \) with a given type \( \kappa \in \mathbb{C} \). The firing of \( t_n \) removes a value corresponding to \( id \) from the place on the left and puts in the place on the right the token \( (id, \kappa) \) where \( \kappa \) is the type of the new object identified by \( id \). This place contains at each time all identifiers in use with their associated types; the actual type associated to an identifier \( id \) may be checked through the action \( \overline{\text{this}}(id, \kappa') \). The transition \( t_d \) represents the object destruction which releases the identifier \( id \).

4.2 Declarations of classes

The \( \text{M-net} \) semantics of a class is mainly composed of an \( \text{M-net} \) for each standard or object attribute and each method. Each of them has again two parts: the resource \( \text{M-net} \) (which is similar to a B(PN)' resource \( \text{M-net} \)) and an interface \( \text{M-net} \) (which is used for handling of dynamic bindings). In order to handle properly the initialisation and the destruction of all the attributes of the instances\(^8\) of a class, the semantics of a class declaration includes also an instantiation and a destruction \( \text{M-net} \). Moreover, if a class \( D \) inherits from a class \( C \), then the semantics of \( D \) contains an additional part composed of request \( \text{M-nets} \) corresponding to all attributes and methods which are not declared in \( D \) but inherited from \( C \) or from an upper class. Each request \( \text{M-net} \) plays the role of a relay allowing to find the right interface and resource \( \text{M-nets} \). Each interface

\(^8\) Note that “all the attributes” of a class also include attributes inherited from its parent class and all upper classes.
M-net is in fact a terminal request M-net directly linked to an attribute or a method resource M-net. See also the schema below.

<table>
<thead>
<tr>
<th>Instantiation handler</th>
<th>Destruction handler</th>
<th>Class C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Att. Resource</td>
<td>Att. Interface</td>
<td></td>
</tr>
<tr>
<td>Meth. Resource</td>
<td>Meth. Interface</td>
<td></td>
</tr>
<tr>
<td>Att. or Meth. Request</td>
<td></td>
<td>For each field inherited and not overloaded in C.</td>
</tr>
</tbody>
</table>

**Attribute M-nets.** The resource M-net of an attribute $a$ keeps the current value of $a$ for each instance of the class where $a$ is declared. If $a$ is a standard attribute, this value belongs to a set $V \subseteq V$ being the type of $a$, and if $a$ is an object attribute, it is a pair $(id, \kappa)$, where $id$ is the identifier of the object of type $\kappa$ assigned to $a$. For instance, if $a$ is an object attribute belonging to a class $C$, then for each object $c$ of type $C$ identified by $id_c$, the resource M-net of $a$ carries a token $(id_c, (id, \kappa))$, where $(id, \kappa)$ is the actual value of the object assigned to $c.a.$

Fig. 3 gives two examples of attribute resource M-nets: $N_{S(id, \kappa, C)}$ for a standard attribute $sa$ of type $V$ in the class $C$ and $N_{O(id_0, D, C)}$ for an object attribute $oa$ of (declared) type $D$ in the class $C$. Initially, in both cases, the places on the left of the attribute resource net contains as many black tokens as there are existing object identifiers, that is $[|\{\}|]$, allowing to store a different value of the attribute for each instance of a class.

![Diagram](attachment://diagram.png)

---

**Fig. 3.** The resource M-nets $N_{S(id, \kappa, C)}$ and $N_{O(id_0, D, C)}$ and the interface M-nets $N_{S\text{Req}(id, C, C)}$ and $N_{O\text{Req}(id_0, C, C)}$ of standard and object attributes $sa$ and $oa$, resp.
Since an attribute can be redefined in a subclass, we distinguish attributes of different classes but having the same name by quoting this name by the class name; this quoting concerns only actions appearing in resource M-nets and their conjugates, e.g., init\_saC, access\_saC and kill\_saC for a standard attribute sa declared in C. The transition \( t_1 \) is used at the creation of an object of the class C identified by \( id_c \) to initialise the attribute saC (resp. oaC). A standard attribute of an object identified by \( id_c \) can then be updated through the action access\_saC(\( id_c, \cdots \)) of the transition \( t_a \) (similarly to what happens for standard B(PN)\(^2\) variables). Updating an object attribute concerns its identifier and its type, namely the pair (\( id_r, \kappa_r \)) with \( r \) for read values, and (\( id_w, \kappa_w \)) with \( w \) for written values. An object attribute may also be explicitly destructed (independently of its owner object) through the transition \( t_d \). Finally, when the owner object is destructed, the associated attributes are also destructed, through the transition \( t_k \). The action \( \text{kill}(id, \kappa) \) triggers the destruction of all attributes of the object identified by the pair \((id, \kappa)\). These destructions are handled by the destruction M-net of the class \( \kappa \), see Fig. 4.

The interface M-net of an attribute \( a \) is intended to allow subclasses to access the right resource when they did not redefine \( a \). Fig. 3 shows the interface M-net \( N_{\text{OReq}(sa,C,C)} \) corresponding to a standard attribute \( sa \) (first argument) of a class C (second argument) declared in C (third argument) and \( N_{\text{OReq}(oa,C,C)} \) corresponding to an object attribute \( oa \) of a class C declared in C. The action \( \text{req\_sa}(id_c, C, \cdots) \) in \( N_{\text{OReq}(sa,C,C)} \) (resp. \( \text{req\_oa}(id_c, C, \cdots) \) in \( N_{\text{OReq}(oa,C,C)} \)) is used for taking into account any access request to the attribute \( sa \) (resp. \( oa \)) of the object identified by \( id_c \).

**Method M-nets.** The resource M-net of a method \( m \) declared in the class C is like a procedure M-net but it takes into account the identifier \( id_c \) of the object to which the method is applied. The standard parameters and local variables are handled as for procedures and object ones also in a very similar way, through the action \( \text{call\_}M_C(id_c, pid, f_1, \ldots, f_n) \), where \( pid \) and \( f_1, \ldots, f_n \) are as for B(PN)\(^2\) procedures, but they may also be object parameters and thus of the form \((id, \kappa)\); \( m \) is quoted here by the class where it is declared, similarly as for attributes. The corresponding interface M-net is then \( N_{\text{MReq}(m,C,C)} \), which is like an interface M-net of a standard attribute, transition \( t_a \) having the label of the form \{ \( \text{req\_}M(id_c, C, f_1, \ldots, f_n) \), \( \text{call\_}M_C(id_c, pid, f_1, \ldots, f_n) \) \}.

**Instantiation M-net.** The creation of a new instance of a class C is the result of the execution of an atomic command comprising the operation new\(_C\)(initlist), see also section 4.3. It involves, in particular, the reservation of a fresh identifier \( id \) in the instances handler M-net \( N_i \) and the initialisation of all the attributes of C for the new instance. This is realised by the instantiation M-net \( N_{\text{Inst}(C, S(sa_1), \ldots, S(sa_k), O(oa_1), \ldots, O(oa_j))} \) of the class C, see Fig. 4, where \( sa_1, \ldots, sa_k \) and \( oa_1, \ldots, oa_j \) are the standard and object attributes of C. The action \( \text{new}(id, C') \) serves to get a fresh identifier \( id \) from \( N_i \), the action \( \text{new\_}C(id', v_1, \ldots, v_l, (id_1, \kappa_1), \ldots, (id_j, \kappa_j)) \) allows to get initial values for all the attributes of C and to send \( id \) back to the expression, and all the actions of the form \( \text{init\_}saC(\ldots) \)
and \( \text{init\_oac}(\ldots) \) are used to initialise the attributes of \( C \). Initially, the place of the instantiation M-net is marked with \([\|]\) black tokens.

\[
\begin{array}{c}
\text{new}(id, C'), \ldots, \text{new}(id, C'), \ldots, \text{init\_saC}(id, v'_i), \ldots, \text{init\_saC}(id, v'_j), \ldots, \text{init\_saC}(id, (id, \kappa_i)!) \}
\end{array}
\]

\[
\text{kill}(null, D) \quad \{ \text{kill}(id, D), \ldots, \text{kill\_attr}(id) \}
\]

\[
\begin{array}{c}
\ldots
\end{array}
\]

\[\text{super}(\kappa, \kappa')\]

**Fig. 4.** The M-nets \( N_{\text{Inst}}(C, a_1, \ldots, a_k, \text{o}a_1, \ldots, \text{o}a_j) \), \( N_{\text{Kill}}(D, a_1, \ldots, a_k) \) and \( N_{\text{H}(\overline{\kappa})} \).

**Destruction M-net.** Each time an instance of a class \( D \) is destructed, the associated attributes are destructed as well. The destruction M-net of \( D \) is intended to handle the destruction of all the attributes of the destructed instance of \( D \).

If \( D \) has no superclass, its destruction M-net is \( N_{\text{Kill}}(D, a_1, \ldots, a_k) \), see Fig. 4, where \( a_1, \ldots, a_k \) are the attributes declared in \( D \). If \( C \) is the superclass of \( D \), then the destruction M-net of \( D \) is \( N_{\text{Kill}}(D, C, a_1, \ldots, a_k) \), which has an additional action \( \text{kill}(id, C) \) in the label of \( t_2 \), which triggers the destruction of all the attributes inherited from \( C \), by the object identified by \((id, D)\). Initially, the place is marked with \([\|]\) black tokens.

**Request M-nets.** Request M-nets are used in a class \( D \) in order to handle correctly all the attributes and methods of \( D \) inherited from a superclass\(^9\) \( C \) and not overloaded in \( D \). They play the role of relays between an access request (to an attribute or a method) and the corresponding interface and resource M-nets, which are actually in the M-nets of \( C \). For instance, if \( \text{o}a \) is an object attribute declared in \( C \) and inherited in \( D \), then the request M-net for \( \text{o}a \) is \( N_{\text{OReq}}(\text{o}a, D, C) \), which is like an object interface M-net, with the transition \( t_a \) having a label of the form \( \{ \text{req\_o}(id, C, (id, \kappa_1)^\top, (id, \kappa_2)!), \ldots, \text{req\_o}(id, C, (id, \kappa_1)^\top, (id, \kappa_2)!), \ldots, \text{req\_o}(id, C, (id, \kappa_1)^\top, (id, \kappa_2)!), \ldots, \text{req\_o}(id, C, (id, \kappa_1)^\top, (id, \kappa_2)!), \ldots \} \). The action \( \text{req\_o}(\ldots) \) will synchronise with its conjugate in the corresponding interface M-net, while \( \text{req\_o}(\ldots) \) will synchronise with its conjugate in the semantics of a command, see also section 4.3. The request M-nets \( N_{\text{SReq}}(\text{o}a, D, C) \) and \( N_{\text{MReq}}(m, D, C) \) for a standard attribute \( \text{o}a \) and for a method \( m \) are defined analogously.

**Inheritance directory.** As explained before, the keyword \( \text{super} \) can be used in order to enforce the access to an attribute or a method of the superclass. Thus, \(^9 \) Note that \( C \) is not necessarily the immediate superclass of \( D \) but is related to it in the inheritance tree.
the class inheritance tree must be known all along the execution. It is ensured by the inheritance directory M-net $N_{H(P)}$, see Fig. 4. The inheritance tree is stored during the execution of the program in the place of $N_{H(P)}$, as the set of tokens $F = \{ (D, C) \mid C$ is the super class of $D \}$. Notice that for allowing concurrent calls to “super” from objects of the same type $D$, the place must contain as many occurrences of token $(D, C)$ as allowed concurrent calls.

### 4.3 Commands

The main commands are composed as in the original B(PN)$^2$, but also allows also object variables and some new expressions at the atomic command level. The semantics of original B(PN)$^2$ commands being unchanged, we address in this section only the semantics of object variables and new kinds of expressions.

**Object variables.** Like a standard variable, an object variable $c$ of class $C$ is represented by an object variable resource M-net $N_{c,C}$ which stores at each time of the execution the pair $(id, \kappa)$ which identifies the object assigned to $c$. At the beginning, this value is initialized to $(null, C)$ and may be updated through the action $\text{access}(c, (id, \kappa))$, which allows to change the current value $(id, \kappa)$ into $(id', \kappa')$. The main difference between an object variable and a standard one is that it may explicitly be destructed, releasing the identifier $id$ used for it. At the resource level, it corresponds to the transition $t_d$, which triggers the destruction of attributes associated to $id$, releases $id$ and sets the value of the variable to $(null, C)$. An example of an object variable resource M-net is given in Fig. 5. Initially, the place contains the token $(null, C)$.

**Method calls and attribute accesses.** Method calls and attribute accesses for an object $c$ are modelled in almost the same way, since the corresponding interface M-nets are very similar. In both cases, we need to know the value $(id, \kappa)$ of $c$ and use it in a request for the corresponding method or attribute. In the case of a standard attribute $sa$ of $c$, the access to $sa$ is modelled by the action allowing to read the value of $c$ and the request action $\text{req}(c, (id, \kappa), x, y)$. The access to an object attribute $oa$ of $c$ is modelled analogously, the request action being $\text{req}(oa, (id, \kappa), (id, \kappa), (id, \kappa))$. For instance, the transition label $(lab)$ of $N_{(lab)}$ being the translation of the atomic BOON command $\langle \text{c, sa > 5} \rangle$, which tests whether the attribute $sa$ of $c$ is greater than 5, becomes $\{ \text{access}(c, (id, \kappa)), (id, \kappa), (id, \kappa)) \} \{ x > 5 \}$.

The call of method $m$ is translated to a call M-net $N_{\text{Call}(lab)}$, where $(lab)$
is the label of the transition devoted to handle \( m \), \( \text{(lab)} \) contains the action \( \text{req}_m(id^i_k, c^i_k, e^i_1, \ldots, e^i_n) \) and all the needed reading actions concerning the effective parameters of \( m \). For instance, the label of the transition handling \( c.m(v, d) \) where \( v \) is a standard variable and \( d \) an object one, becomes:

\[
\{ \text{access}_c(id, \kappa), (id, \kappa) \}, \text{access}_v(x, x'), \text{access}_d((id_d, \kappa_d), (id_d, \kappa_d)') \}, \text{req}_m((id^i_k, c^i_k, x), (id_d, \kappa_d)) \},
\]

while that of the method call \( c.a.m(5) \), where \( a \) is an object attribute of \( c \), and \( m \) is a method of \( a \) is:

\[
\{ \text{access}_c((id_c, \kappa_c), (id_c, \kappa_c)), \text{req}_m(id^i_k, \kappa^i_k, 5'), \text{req}_a(id^i_k, \kappa^i_k, (id_d, \kappa_d), (id_d, \kappa_d)) \}.
\]

**Assignments.** Assignments to object or standard variables or attributes are modelled in a similar way as for \( B(PN)^2 \) variables. For an attribute \( a \), it is realised through the action \( \text{req}_a \), while for a variable \( c \), through the action \( \text{access}_c \). The assignment of an object variable (resp. an object attribute) consists in updating the identifier of the variable (resp. the object attribute) and its type. For instance, the label of the one-transition M-net corresponding to the command \( (c.a' = 5) \) is \( \{ \text{access}_c((id_c, \kappa_c), (id_c, \kappa_c)), \text{req}_a(id^i_k, \kappa^i_k, x, 5') \} \).

**Object creation and destruction.** An object is usually assigned to an object variable or attribute, otherwise it is not reachable. Thus, the class instantiation and object destruction are associated to a variable or attribute access.

**Class instantiation.** The semantics of a class instantiation \( \text{new}_C(\text{initial list}) \) consists in creating a new object of the class \( C \) and in initialising all the attributes with the corresponding values, through the action \( \text{new}_C(id, v^i_1, \ldots, v^i_j, (id_1, \kappa_1), \ldots, (id_j, \kappa_j)) \), in order to import a fresh object identifier \( id \) and to export the initial values of the attributes of the new object. For instance, the label of the one-transition M-net corresponding to the command \( (c' = \text{new}_C(5, d)) \) where \( c \) and \( d \) are object variables is:

\[
\{ \text{access}_c((id, \kappa), (id, C)'), \text{access}_d((id_d, \kappa_d), (id_d, \kappa_d)'), \text{new}_C(id, 5'), \text{new}_C(id, \kappa_1)' \}.
\]

**Object destruction.** The destruction of an object \( \text{del}(c) \) consists in releasing its identifier and destroying the corresponding instance of all its attributes. The destruction of attributes is handled by the destruction M-net of each class. The releasing of the identifier of \( c \) is managed by its resource M-net. Thus, for instance, the M-net modelling the atomic BOON command \( (\text{del}(c)) \) is just a one-transition M-net with the label \( \{ \text{del}_c \} \). For the explicit destruction of an object attribute \( c.a \), the label of the corresponding M-net is \( \{ \text{access}_c((id_c, \kappa_c), (id_c, \kappa_c)'), \text{reqdel}_a(id^i_k, \kappa^i_k)' \} \).

**The keywords “super” and “this”.** The translation of a command involving an attribute or a method of a superclass, represented by the presence of strings \( \text{super}^j \), generates in the labels of corresponding transitions the additional actions \( \text{super}(\kappa_0, \kappa_{s}), \ldots, \text{super}(\kappa_{s-1}, \kappa_{s}) \). For instance, the transition label corresponding to \( (c \text{super} \text{super}, \text{sa} > 5) \) is \( \{ \text{access}_c((id_c, \kappa_c), (id_c, \kappa_c)'), \text{req} \text{sa}(id^i_k, \kappa^i_k, x, x'), \text{super}((\kappa^i_k, \kappa^i_1), \text{super}(\kappa^i_1, \kappa^i_2)) \} \{ x > 5 \} \).

In a method body, the keyword “this” refers to the object on which the method is applied. Since at its execution, a method carries the identifier of the
object on which it is applied, we get the type of this object through the action
\(this(id, \kappa)\), where \(\kappa\) is to be sent by the class instances M-net \(N_t\). For instance,
the transition label corresponding to the command \(< c.a' = this >\) in the
method applied on an object identified by \(id\) is: \(\{this(id^1, \kappa), \text{access}_C((id_c, \kappa_c),\)
\((id_e, \kappa_e))\}, \text{req}\_a(id^1_c, \kappa^1_c, (id_e, \kappa_e), (id, \kappa^1))\}\).

4.4 Soundness of the solution

In this section we bring together some arguments allowing to support the sound-
ness of our semantics. This concerns mainly the class instantiation, object ini-
tialisation, inclusion polymorphism and dynamic bindings.

Actually, as explained in section 4.3, the operation \(\text{new}_C(\text{initlist})\) allows to
create a new instance of the class \(C\) and to initialise all the attributes according
to "initlist". The new instance is uniquely identified by \(id\) coming from the
class instances M-net \(N_t\). More precisely, getting \(id\) is relayed, using the action
\(\text{new}_C(id, v^1, \ldots)\), by the instantiation M-net \(N_{\text{Inst}}(C, S(a_1), \ldots, O(a_n))\). It
gets \(id\) from the M-net \(N_t\) using the action \(\text{new}(id, C)\) and initialises all the
attributes of the new object with corresponding initial values through actions
\(\text{init}\_a(id, v^1)\) if \(a\) is a standard attribute and \(v^1\) is its initial value (resp.
\(\text{init}\_a(id, (id_{a_i}, \kappa_{a_i})), (id_{a_i}, \kappa_{a_i}))\) if \(a_i\) is an object attribute). The object creation is sound
because \(N_t\) does not allow to create two distinct objects with the same identi-
der and \(N_{\text{Inst}}(C, S(a_1), \ldots, O(a_n))\) correctly initialises all attribute resource M-nets.
During the execution, the attributes and the methods are always supplied with the
identifier of the object they correspond to. Each attribute access and each
method call takes the object identifier into account that guarantees a correct
handling of them.

The proposed extension supports the inclusion polymorphism, it means that
it is possible to assign an object of a subtype of the declared type to an object
variable or an object attribute. In such a case, the right attributes and methods
must be found at the execution time (dynamic binding), as described in section 4.
This is obtained using “relay” M-nets (namely, the request and interface M-nets)
between the attribute or method invocation and the corresponding resource M-
nets. The inheritance tree is known statically which allows to build correctly the
request M-nets w.r.t the corresponding interface M-nets, which is crucial and
allows to handle the right attribute or method. An analogous mechanism, but
working in the opposite direction, is used in order to access the attributes or
methods of the upper classes when the keyword \text{super} is used.

5 Conclusion

We introduced a new formalism, called BOON (Basic Object-Oriented Nota-
tion), devoted to specify concurrent systems using object-oriented concepts. We
defined for it a fully compositional Petri net semantics in terms of M-nets (a
class of coloured Petri nets provided with a process algebra-like operations).
This led to the development of a representation for classes, objects, attributes
and methods supporting single inheritance and dynamic bindings.

This approach may easily be extended in order to allow for more polymorphism, for instance, by accepting several user-defined constructors or different methods having the same name. The former can be modelled in our approach using customised instantiation M-nets, while the latter can be obtained by simply preprocessing the homonymous methods in order to give them distinct names.

Our future work will consist in enriching BOON with encapsulation features allowing to give to class fields various access restriction levels (private, public,...). Our interest will also focus on the application of M-nets to the translation of UML [4] diagrams.

References
Appendix

This appendix gives an example a BOON program $Prog$, described below, and its translation into M-nets.

```
program class C {
    att \( a_1 : \{1, 2, 3\} \)
    method \( m() \) begin ... end
}
class D : C {
    att \( a_2 : \{true, false\} \)
    method \( m() \) begin ... end
}
begin over \( c : C \) \( \langle c' = new_C(1, false) \rangle \); \( c.m() \) end
```

The translation of the above program $Prog$ is the initially marked (with one token • in each entry place) M-net:

```
Mnet \( \langle Prog \rangle = \) (\( N_1 \) || \( N_{H(D,C)} \))
|| ( \( N_{\text{Inst}(C,S(a_1))} \) || \( N_{KILL}(C,\bot,S(a_1)) \)
    || \( N_{S(a_1,\{1,2,3\},C)} \) || \( N_{\text{req}(a_1,C,C)} \)
    || \( N_{M(m,C)} \) || \( N_{\text{req}(m,C,C)} \))
|| ( \( N_{\text{Inst}(D,S(a_2))} \) || \( N_{KILL}(C,\bot,S(a_2)) \)
    || \( N_{S(a_2,\{true,false\},C)} \) || \( N_{\text{req}(a_2,D,D)} \)
    || \( N_{M(m,D)} \) || \( N_{\text{req}(m,D,D)} \)
    || \( N_{\text{req}(a_1,D,C)} \))
|| ( \( N_{c,C} \))
|| ( \( N_{\text{access}_c((id,\kappa),(id,C))}, new_C(id,1,\bot) \}) ;
    \( N_{\text{Call}}(\text{access}_c((id,\kappa),(id,\kappa)'),\text{req}_m(id,\kappa)) \))
```

\( scA \),

where \( A \) is the set of actions involved in the M-nets composing $Mnet(Prog)$, e.g.,
new, del, this, super, new$C$, init$\_a1_C$, access$\_a1_C$, kill$\_a1_C$, req$\_a1$...