An Object Oriented Specification Language based on Hierarchical Algebraic Petri Nets*

Olivier Biberstein
CUI, University of Geneva
CH-1211 Genève 4, Switzerland
biberste@cui.unige.ch

Didier Buchs
EPFL, DI-LGL
CH-1015 Lausanne, Switzerland
buchs@di.epfl.ch

Abstract

During the last few years there has been significant interest in combining the benefits of an object-oriented approach with Petri net formalism which is suitable to modelling and simulation of concurrent systems. Nevertheless, for modelling some system behaviour dynamic creation of new components is often needed.

In this paper we introduce a formalism based on algebraic Petri nets using the object-orientation paradigm. Object classes are specified by means of algebraic Petri nets, synchronization expressions link the class instances and static as well as dynamic creation of objects is allowed.

Inheritance and subtyping are considered as two different mechanisms. Inheritance is viewed as a duplication mechanism while subtyping is considered at a semantic level as a relation between models. The order-sorted algebras framework captures the subtyping notion between the objects which are identified by order-sorted values.

Language semantics is defined through distributed transition systems over the models of algebraic specification.

1 Introduction

During the last few years we have developed, both theoretically [BG91] and for modelling of concurrent languages [BG93], a formal object-based specification language built on algebraic specifications and Petri nets called CO-OPN (Concurrent Object Oriented Petri Nets).

CO-OPN has been developed in order to offer a formal specification language for the design of large parallel systems. The CO-OPN objects are interconnected modular entities described by algebraic nets which are Petri nets combined with algebraic abstract data types. Although algebraic nets are an important improvement of Petri nets, they are quite unsuitable when faced with large problems. Therefore, some facilities have been included in CO-OPN such as structuring capabilities through synchronization mechanisms between objects, hierarchical algebraic specifications and genericity both on data types and objects. Tools such as a graphical editor, a compiler and a simulator for prototyping have been built and are packaged in SANDS (Structured Algebraic Net Development System) [BFR93].

The formal semantics of CO-OPN have recently been generalized in the definition of Hierarchical Algebraic Petri nets (HAP nets) [Gue94]. This work overcomes the previous limitation of CO-OPN regarding the hierarchical dependencies between the objects. Viz. in HAP nets, cyclic synchronizations between objects are allowed.

In this paper, we propose a new language, called CO-OPN/2, which improves CO-OPN by introducing the still missing or relatively poorly integrated object-oriented features (i.e. class notion, subtyping, inheritance). An object corresponds to an instance of a class, each class is described by an algebraic Petri net and each instance is identified by values of order-sorted algebras. The framework of order-sorted algebras [GM89], already used for the purpose of object oriented programming semantics [Bre93], captures the notion of subtyping between objects, whereas the one of algebraic Petri nets provides the true concurrency features.

In Petri nets and other models including concurrency, a major interest was to model object-based systems where inheritance was considered as a duplication of the behavior description and no polymorphism notion was provided. In our approach, we differentiate subtyping and inheritance as recommended in [Sny86]. Inheritance is still considered as a duplication mechanism whereas subtyping is viewed at a

*This work is supported by the FNRS (Fonds de la Recherche Scientifique Suisse). Project 21.32286.91

semantic level as a relation between models. A subclass inherits of a superclass means that the subclass "reuses" the components of the superclass. Therefore, the two classes induce two differents classes of models. However, during subtyping, the two sets of models are related, the superclass set of models "includes" all the subclass models where inclusion means the possible behavioral substitution.

The semantic aspects of the CO-OPN/2 language are supported by the HAP nets and the order-sorted algebras. Particular problems such as class semantics and self reference are modeled in terms of these frameworks. Not explained in this paper, the refinement of both the classes and the algebraic abstract data types has been defined and is based on observational equivalence and bisimulation, as already developed in the theory of CO-OPN and HAP nets.

Many other works are investigate in the direction of concurrent object-oriented specifications. Among these studies we may mention the general models of concurrency including data structures, algebraic specification of concurrency [AR93], dynamic data types [Reg93] and objects and temporal logic [FM90]. In the particular field of Petri nets related with HAP nets, we have High Level Petri nets [JR91], Algebraic Nets [Vau87, Rei91], Structured Algebraic nets [BCM88]. More recently, some extensions of Petri nets including object-orientation features have appeared [BP90, BC93, Lak94, SB94].

The paper is organized as follows. In the next section we introduce the CO-OPN/2 syntax by means of an example. Then, Hierarchical Algebraic Petri nets and a formal definition of the abstract syntax are presented in section 3. Finally, the section 4 gives the semantics of the language with Hap nets.

2 An informal example

This section introduces the syntactic aspects of the specification language CO-OPN/2 through an informal example. To present the language we have chosen the well known Dijkstra's philosopher example. The context is still one of philosophers sitting around a table with one fork lying between two philosophers. The philosophers are in the thinking state and to move into the eating state each of them needs to hold their left and right forks.

The basic objects in our example are a 'restaurant' with five 'seats' around a 'table', five 'forks' on it and some 'philosophers' who can enter the restaurant, sit at the table and finally leave. If all the free seats are occupied new philosophers will either have to wait for someone to leave or a new seat and an additional fork can be added. Once seated a philosopher can take his left and right forks and enter the eating state then return to his previous state.

To express object oriented features we have introduced three hierarchically organized kinds of philosophers. For the example, philosophers at the top of the hierarchy belong to the Socrates-Plato-Aristotle philosophy school and are simply called philosophers. From this school two different trends which do not share the same ideas appeared: stoicism and epicurism. Moreover, objects can refer their methods (self-reference) as well as dynamically create other objects. For example it is possible to add a new seat and a new fork at any time on the table or a philosopher can be dynamically created and arrive in the restaurant.

2.1 Structure of the language

A specification in CO-OPN/2 is composed of two different modules. One is called *specification module* (`Spec Module`) which is used to describe algebraic abstract data types (ADT) and the other one is called *class module* (`Class Module`) which expresses the behaviour of a set of objects by means of algebraic Petri nets (APN). A class can be considered simply as an template object and objects as class instances. We will show that class instances can be created statically as well as dynamically.

To be as close as possible to an actual situation every relevant "material" component of the dcor is considered as an independent object while "immaterial" things correspond to algebraic data structures. In our example, natural numbers and blacktoken are immaterial. Then specification modules are used to specify these data structures as shown in frame 1.

A specification module is composed of two major parts: a signature and a body. A module signature consists of all the components of an ADT we want to be visible from the outside: sorts, generators, operations, subsorts relations and used modules. The body section includes the properties of the operations by means of axioms but also all the private things needed by the ADT such as sorts, generators, operations, variables and private used modules. Other features are not illustred here in order not to overload the example. Among them we can mention: module genericity coupled with morphisms and renamings, order relation over sorts and nested modules for better structure.

Operation notation is called *midfix* (for prefix, postfix, infix). In the **Operations** field of frame 1 one sees how to declare an operation and its profile, underscore is used to show where the parameters are and the following arity gives the sorts (corresponding clearly from left to right). In the expressions, applicative form is adopted when parameters are applied to an operation. Thus, parentheses are only needed when ambiguity appears.

Class modules have the same two parts (signature, body) as specification modules. Their meaning is the same but the content differs a bit. The signature section contains four fields. As previously mentioned all used specification modules but also used class modules are listed in a **Use** field. The type of the class instances has to be given explicitly and when inheritance is desired subtyping relation superclass and subclass must be given in the **Ordering** field. Methods are grouped in the signature under the **Methods** field. The same midfix notation is used for the methods except that they do not return any result since they are used during synchronizations. Finally, we also find in the signature part the static instance definitions in the object field as we will see in the restaurant and table class.

The body part comprises transitions and places of a classic APN. Synchronizations and internal transitions are not incompatible.

```
Spec Module Basic;
Signature
 Sorts bool, nat;
 Generators
 true, false : bool;
 0     : nat;
 s _   : nat → nat;
 Operations
 - _   : bool → bool;
 _+_   : bool bool → bool;
 _+_   : nat nat → nat;
 _-_   : nat nat → nat;
 _==_ : nat nat → bool;
 _<_   : nat nat → bool;
  ... other standard operations
Body
 Axioms
 -true   = false;
 -false  = true;
 true+b  = true;
 false+b = b;
 n+0 = n;
 0+(s n) = s n;
 (s n)+(s m) = s(s(n+m));
  ... and their associated axioms
 Where
 b : bool;
 n, m : nat;
End Basic;

Spec Module Blacktoken;
Signature
 Sorts blacktoken;
 Generators token : blacktoken;
End Blacktoken;
```

Frame 1 : Basic algebraic abstract data types

The difference with the methods is that internal transitions can be fired at any time (as long as pre/post-conditions are respected) whereas methods can only be performed when all internal transitions have been fired.

As in CO-OPN, a multiset constructor (the braces) is provided for the places. In CO-OPN/2, a place becomes now a partial function (dynamically defined) or in other words a place can be considered as an array of values with the undefined value. We will call this kind of place an association. This provides the possibility to "consult" the place content without taking out any value. The semantics of this improvement seems natural but is not explained here.

The associated axioms of the transitions/methods describe the properties of the specification events. The keyword **With** allows to express the synchronizations with other partners. Multiple synchronizations can be performed simultaneously, sequentially or alternatively with the respective operators '&', '..' and '+'. The pre-conditions (separated by comas) and the post-conditions are respectively on the left and right side of the symbol '→'.

Each object has a special method, called **create**, which takes charge of the creation of a new instance and its initialization. The axioms located in the **Initial** field describe this initialization. Some parameters can be given for this initialization as we will see later.

2.2 Fork and seat classes

The two simplest actual objects of our example are forks and seats. Although they do not have any own behaviour and, therefore, could be easily represented by means of ADT, we have chosen to modelize them with APN since they are material entities. Frame 2 shows the classes of these two objects with their respective type declaration; these objects do not have any methods, places or transitions and the body section is left blank.

```
Class Module Fork;
Signature
 Type fork;
End Fork;

Class Module Seat;
Signature
 Type seat;
End Seat;
```

Frame 2 : Fork and Seat object classes

2.3 Restaurant class

Let us introduce a more interesting partner: the restaurant (frame 3). Although instances of a class can dynamically be created, only one restaurant is present in the example. We call this instance `Zorba's` and statically define it in the field **`Objects`**. The philosophers "arrive" through the parametized methods `arrive` and go to the bar as one sees in the axiom of this method. The place `bar` stores the new philosophers waiting for an available seat around the table while `cashdesk` represents sat philosophers before leaving the restaurant. Each of these two places is a multiset (the braces correspond to the multiset constructor) and the multiset values are themselves objects (philosophers) or more precisely references to philosopher objects. When a seat is not occupied, a philosopher can move from the bar to the table through the internal method `takesit`. In the associated axiom one observes a synchronization with a method `sitdown` of an object named `tbl` corresponding to the table (the usual dot notation is used when calling a method). Thus a philosopher `p` is taken from the place `bar` and moved to the table (frame 7).

```
Class Module Restaurant,
Signature
 Use Table, Philosopher;
 Type restaurant;
 Objects Zorba's : restaurant;
 Methods
 arrive _ : philosopher;
 leave;
Body
 Transitions
 takesit;
 gopay;
 Places
 bar      : → {philosopher};
 cashdesk : → {philosopher};
 Axioms
 arrive p :: → bar{p};
 leave    :: cashdesk{p} → ;
 takesit With tbl.sitdown p:: bar{p}→;
 gopay   With tbl.standup p::
         → cashdesk{p};
 Where  p : philosopher;
End Restaurant;
```

Frame 3 : The restaurant (at Zorba's)

2.4 Philosopher class

Philosopher objects (frame 4) have many functionalities. The two methods `goeat` and `gothink` allow philosophers to change from the eating state to the thinking state and vice versa. In our specification the `tbl` table knows where each philosopher is sat and where their right/left side fork are. Thus, the table gives some services to the outside world as we are going to see in frame 7. For a deeper description of these services, one can have a look in the table class and its associated explanations which are intentionally left at the end of this section. In the axioms of the `goeat` method, the synchronization `tbl.takeforks f f'` **`self`** "asks" to the `tbl` object the left/right side fork `f` and `f'` of oneself (the special identifier **`self`** is the object reference onto oneself). Of course, if an immediate neighbour is already using his own forks the synchronization can not be performed.

```
Class Module Philosopher;
Signature
 Type philosopher;
 Methods
 goeat; gothink;
Body
 Use Fork, Table, Blacktoken;
 Places
 forks : → {fork};
 eat, think : → {blacktoken};
 Axioms
 goeat With tbl.takeforks self f f' ::
  think{t} → eat{t}, forks{f,f'};
 gothink With tbl.release f f' ::
  forks{f,f'}, eat{t} → think{t};
 Where
 t : blacktoken;  f, f' : fork;
 Initial
 create :: → think{token};
End Philosopher;
```

Frame 4 : Philosopher object class

In case of success, the philosopher represented by the token `t` is then moved from the place corresponding to the thinking state into the eating state and his two forks are stored in the place `forks`. At this time the philosopher is in an eating state and his two forks are no longer on the table but in the place `forks`. The `gothink` is quite similar but returns the forks to the table and moves the philosopher from the eating to the thinking state.

As already mentioned, the **`Initial`** field describes how new instances are initialized. Thus, new philosophers are automatically put in a thinking state by the **`create`** axiom.

2.5 Philosophers subclasses

```
Class Module Stoic;
Signature
 Use Philosopher;
 Type stoic;
 Ordering stoic < philosopher;
 Methods godrinkwater; stopdrink;
Body
 Places
 drinkwater : → {blacktoken};
 Axioms
 godrinkwater ::
  think{t'} → drinkwater{t'};
 stopdrink ::
  drinkwater{t'} → think{t'};
 Where t' : blacktoken;
End Stoic;
```

Frame 5 : Stoic class (heir of Philosopher)

```
Class Module Epicurean;
Signature
 Use Philosopher;
 Type epicurean;
 Ordering epicurean < philosopher;
 Methods  godrinkwine; stopdrink;
Body
 Places
 drinkwine : → {blacktoken};
 Axioms
 godrinkwine ::
  think{t'} → drinkwine{t'};
 stopdrink ::
  drinkwine{t'} → think{t'};
 Where t' : blacktoken;
End Epicurean;
```

Frame 6 : Epicurean class (heir of Philosopher)

Inheritance mecanism is illustred by two frames (frame 5 and frame 6). Let us observe the signature of frame 5. The **Use**, **Type** and **Ordering** fields mean that the new class `Stoic` inherits all the characteristics of Philosopher class. This also means that instances of the new class behave in the very same way than philosopher instances and, in this case, we say that `stoic` is a *subtype* of `philosopher` (we write `stoic < philosopher`). This subtype relation enables us to affirm that an object of type `stoic` is also of type `philosopher` and that any object of type `philosopher` can be substituted by any object having the type `stoic`.

Stoic class inherits the behaviour described in the Philosopher class, but adds a new behaviour by means of the two methods `godrinkwater` and `stopdrink`. These new methods and their associated axioms do not interfere with the original behaviour, as stoic object are just allowed to move from a thinking state to a drinking water state and return to the previous state. The same reasoning could be applied to for the Epicurean class.

By now, our example is built with a hierarchy of three kinds of philosophers. Recall the place `bar:→{philosopher}` in the restaurant where philosophers are waiting for an available seat. This place can now contain original philosophers as well as stoic or epicurean philosophers. This captures the notion of polymorphism. In the next class we will see a more sophisticated example concerning polymorphism.

2.6 Table class

This is the most complex class. First, let us have a look at the places field in order to know what kind of data or objects a table contains. The fourth first places are associations to determine what the left and right fork or seat of another seat is. The fifth provides on which seat a philosopher is sat. The places `freeseats` and `busyseats` are multisets of seats where a seat changes from free to busy state. `forks` is a multiset of forks (recall that a philosopher “asking” his forks “removes” them from this multiset to put them is their own place). `takeforks` and `release` are methods called by philosophers when they want to move from one state to the other one while `standup` and `sitdown` are called by the restaurant.

The method `sitdown` is particularly interesting because it modelizes the fact that a new stoic philosopher can only sit down next to two philosophers of the same type as his or his supertype. The first axiom associated to the `sitdown` method solves the problem for stoic philosophers. Two neighbours `(e,e')` are selected and the inequalities `p≠e` and `p≠e'` avoid epicurean philosophers. The next axiom solves the problem for epicurean philosophers in a similar manner while the third one allows philosopher at the top of the hierarchy (neither stoic nor epicurean) to sit down next to any kind of philosopher.

The last remark is related to the creation and the initialization of table objects. The creation process can be parametrized as in the table class. Formal parameters types are given in the **Type** field next to the type identifier while effective parameters can be given when static or dynamic creation of a new object is invoked. In the table class, only static creation of the object `tbl` is done with 4 as effective parameter value. This parameter corresponds to the formal argument `n` in the **create** axioms and it is used to create the right number of seats and forks. This time the **create** axioms are recursive. The basic case **create** `0` takes charge of the creation of the first seat and fork and updates the associations, while the recursive case sequentially calls `n` times the `newseat` method of the class itself. This method `newseat` selects two seats `s'` and `s"`, creates a new seat and a new fork and inserts the two new objects between the two selected seats updating the associations.

```
Class Module Table;
Signature
 Use Basic, Fork, Philosopher;
 Type  table _ : nat;
 Objects tbl 4 : table;
 Methods
 takeforks _ _ _ : philosopher fork fork;  release _ _ : fork fork;
 standup _ ,  sitdown _ : philosopher;
 newseat;
Body
 Use Seat, Epicurean, Stoic;
 Places
 leftfork  _ : seat → fork;  rightfork _ : seat → fork;
 leftseat  _ : seat → seat;  rightseat _ : seat → seat;
 philoseat _ : philosopher → seat;
 freeseats   : → {seat};   busyseats   : → {seat};
 forks       : → {fork};
 Axioms
 takeforks p f f' :: s=philoseat p, f=leftfork s, f'=rightfork s, forks{f,f'} → ;
 release f f' :: → forks{f,f'};
 sitdown p :: s'=leftseat s, s"=rightseat s, s'=philoseat e, s"=philoseat e',
  p≠e, p≠e', freeseats{s} → busyseats{s}, philoseat p:=s;
 sitdown p :: s'=leftseat s, s"=rightseat s, s'=philoseat c, s"=philoseat c',
  p≠c, p≠c', freeseats{s} → busyseats{s}, philoseat p:=s;
 sitdown p :: p≠c, p≠e, freeseats{s} → busyseats{s}, philoseat p:=s;
 standup p :: s=philoseat p, busyseats{s} → freeseats{s}, philoseat s:=⊥ ;
 newseat With s.create & f.create :: s'=leftseat s" →
  forks{f}, freeseats{s}, rightseat s':=s, leftseat s":=s, leftseat s:=s',
  rightseat s:=s", rightfork s':=f, leftfork s:=f,  rightfork s:=leftfork s";
 Where
 p,p':philosopher; c,c':stoic; e,e':epicurean;  s,s',s":seat; f,f':fork; n:nat;
 Initial
 create 0 With f.create & s.create :: → forks{f}, freeseats{s},
 leftfork s:=f, rightfork s:=f, leftseat s:=s, rightseat s:=s;
 create (s n) With create n .. self.newseat;
End Table;
```

Frame 7 : Table object class

Figure 1 Partial overview of the relationships between classes.

3 Basic notions of structured algebraic Petri nets

This section introduces the basic aspects of the Hierarchical Algebraic Petri nets (HAP-nets) which are used to describe the semantics of the CO-OPN language. The project CO-OPN has been developed in order to propose a formal specification language for the design of large parallel systems. This aim is reached by using modular notions with an algebraic specification language combined with Petri Nets. More precisely, the CO-OPN objects are interconnected modular entities described by algebraic nets. In fact, CO-OPN is based on two simple and elegant concepts, algebraic data types [EM85] and Petri nets [Rei85], gathered into a more general model: algebraic nets [Vau87, Rei91]. Although algebraic nets are an important improvement of Petri nets, they are quite unsuitable when faced large problems. Therefore, some facilities have been included in the CO-OPN model such as genericity, standard constructors on data types and structuring capabilities. Subtyping, a major object-oriented feature missing in CO-OPN, is introduced in CO-OPN/2 by means of order-sorted algebra.

3.1 Order-sorted algebraic specifications

An abstract data type is the representation of a data structure coupled with a set of operations that manipulate this data structure (data type) independently of any particular implementation (abstract). The algebraic approach to specification considers a system in terms of abstract data types. Mathematically speaking, these data types are represented by many-sorted algebras together with axioms or equations where an algebra is simply a domain provided with a set of operations.

We consider the given universe U which includes the following disjoint sets: the set of all sort names $SORTS$ the set of all function names $FUNC$ and the set of all variable names VAR.

3.1.1 Order-sorted signature, order-sorted algebra

Definition 3.1 *Order-sorted signature*
An *order-sorted signature* $\Sigma = (S, \leq, F)$ is a partial order $\leq$, a set of sorts $S \subseteq SORTS$ and an $S^* \times S$ indexed family $F = (F_{w,s})_{w \in S^*, s \in S}$ with $F \subseteq FUNC$ of operations names f. The set F satisfies the *monotonicity* condition: if $f \in F_{w_1,s_1} \cap F_{w_2,s_2}$ and $w_1 \leq w_2$ then $s_1 \leq s_2$. When $f \in F_{w,s}$ we say that w is the *arity*, s the *coarity* and $\langle w, s\rangle$ the *rank* of the function. Sometimes we denote by $f \in F_{s_1 \ldots s_n, s}$ by $f : s_1 \ldots s_n \longrightarrow s$.

Some properties are required on the signature such as regularity and coherence.

Definition 3.2 *regular and coherent signature*
A signature $\Sigma = (S, \leq, F)$ is *regular* iff given a function $f \in F_{w_1,s_1}$ and $w_0 \leq w_1$, there exists is a least rank $\langle w, s\rangle \in S^* \times S$ such that $f \in F_{w,s}$ and $w_0 \leq w$. A signature $\Sigma = (S, \leq, F)$ is *coherent* iff Σ is regular and for each sort $s \in S$ has a maximum $(\max(s))$ in S, i.e. $s \leq \max(s)$ and $s \leq s'$ implies $s' \leq \max(s)$ for all $s' \in S$.

The models are Σ-algebras that satisfies the order-sorted constraint. In an intuitive way, a subsort deals with less value and has less defined functions than its super sort.

Definition 3.3 *Order-sorted algebra*
Let $\Sigma = (S, \leq, F)$ be a signature. A partial order-sorted Σ-algebra consists of an S-indexed set $A = (A_s)_{s \in S}$ and an F-indexed family $F^A = (f^A_{s_1 \ldots s_n s})_{f : s_1 \ldots s_n \rightarrow s \in F}$ where $f^A_{s_1 \ldots s_n s}$ is a function from $A_{s_1} \times \ldots \times A_{s_n} \longrightarrow A_s$ such as:

1) $s \leq t \in S$ implies $A_s \subseteq A_t$
2) $f \in F_{s_1 \ldots s_n, s} \cap F_{t_1 \ldots t_n, t}$ with $(s_1 \ldots s_n, s) \leq (t_1 \ldots t_n, t)$ implies $f^A_{s_1 \ldots s_n s} = f^A_{t_1 \ldots t_n t} \mid_{A_{s_1} \times \ldots \times A_{s_n}}$ i.e. $f^A_{s_1 \ldots s_n s}(a_1, \ldots, a_n) = f^A_{t_1 \ldots t_n t}(a_1, \ldots, a_n)$ for all $a_i \in A_{s_i}$ $i = 1, \ldots, n$.

We usually omit the family F^A and write A for a partial order-sorted Σ-algebra (A, F^A). Moreover we denote the set of all Σ-algebras by $Alg(\Sigma)$.

Definition 3.4 *order-sorted Σ-homomorphism*
Let $\Sigma = (S, \leq, F)$ a signature, $A, B \in Alg(\Sigma)$ two Σ-algebras. An order sorted Σ-homomorphism $\mu : A \longrightarrow B$ is a collection of partial functions $(\mu_s : A_s \longrightarrow B_s)_{s \in S}$ such that

1) $\mu_s(f^A_{s_1 \ldots s_n}(a_1, \ldots, a_n)) = f^B_{s_1 \ldots s_n}(\mu_{s_1}(a_1), \ldots, \mu_{s_n}(a_n))$
2) $s \leq t$ implies $\mu_s(a) = \mu_t(a)$ for all $a \in A_s$.

3.1.2 Terms, axioms and presentations

The operation properties are expressed by means of *equations* or *axioms* which are a couple of well formed terms over a signature.

Definition 3.5 *Terms*
Let $\Sigma = (S, \leq, F)$ be a signature and X an S-indexed family of disjoint sets of variables. The S-indexed $((T_{\Sigma,X})_s)_{s \in S}$ set of all terms over a signature Σ and a set of variables X is the smallest set defined inductively as follows:

1) $\forall x \in X_s \quad x \in (T_{\Sigma,X})_s$,
2) $\forall f : \rightarrow s \in F \quad f \in (T_{\Sigma,X})_s$,
3) $\forall (t_1, t_2, \ldots, t_n) \in (T_{\Sigma,X})_{s_1} \times (T_{\Sigma,X})_{s_2} \times \ldots \times (T_{\Sigma,X})_{s_n}$,
$\forall f : s_1, s_2, \ldots, s_n \rightarrow s \quad f(t_1, t_2, \ldots, t_n) \in (T_{\Sigma,X})_s$,

Since a term may have many types, the interpretation in some algebra leads to ambiguities. However, there is a proposition saying that for coherent signatures each term can be associated with a unique sort (least).

For a Σ-algebra A, an *assignment* σ is an S-morphism (a morphism which respects the sorts) from the set of variables X into A.

Definition 3.6 *Interpretation*
Let Σ be a signature, A be a Σ-algebra and σ be an assignment from a set of variables X into A, an *interpretation* in A is a Σ-morphism μ_σ from $T_{\Sigma,X}$ into A such as:

1. $\mu_\sigma(cst) = cst_A$,
2. $\mu_\sigma(f(t_1, \ldots, t_n)) = f_A(\mu_\sigma(t_1), \ldots, \mu_\sigma(t_n))$,
3. $\mu_\sigma(x) = \sigma(x)$ where $x \in X$.

Definition 3.7 *Axiom and validity*
An axiom is said *positive conditional* if and only if $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \Rightarrow t = t'$ where $t_i, t'_i \in (T_{\Sigma,X})_{s_i}$ and $t, t' \in (T_{\Sigma,X})_s$. A positive conditional axiom $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n \Rightarrow t = t'$ is *valid* in a Σ-algebra A if for any assignment σ, for all $i = 1, \ldots, n$, $\mu_\sigma(t_i) = \mu_\sigma(t'_i)$ then $\mu_\sigma(t) = \mu_\sigma(t')$.

Definition 3.8 *Specification*
Given a set of variables X, a set of positive conditional axioms Ax and a signature (S, F), a *specification* is a 5-uple $SPEC = (S, \leq, F, X, Ax)$.

Definition 3.9 *Validity of a specification*
Let $SPEC = (S, \leq, F, X, Ax)$ be a specification. We say that *SPEC* is *valid* in a Σ-algebra A when every axiom in *SPEC* is valid in A. *Alg(SPEC)* denotes the subclass of $Alg(\Sigma)$ where *SPEC* is valid.

3.2 Algebraic nets

Algebraic nets are a powerful combination of Petri nets, modelling the concurrency or the dynamic part, and the algebraic abstract data types modelling the data structure part of a specification. Petri nets are used to model and analyze discrete systems in which notions such as interacting components, concurrency and synchronization play a key role. Petri net modelling embeds both the structural aspect of the system and its evolution. Systems are modelled through the use of places and transitions. Their behaviour is described using rules including pre- and post-conditions on the states. The use of tokens with an internal structure greatly increases the modelling power of Petri nets. Structured tokens give an insight into the way data structures evolved during the system running and many aspects of the modelling are described in a more accurate and concise manner. The evolution of the structured tokens are specified by rules of the underlying formalism used to represent these tokens. By its abstractness and soundness, algebraic specifications turn out to be well-suited for this purpose. The high-level nets obtained by describing structured tokens in the algebraic framework are called algebraic nets.

Although we introduce here the Algebraic nets in the same way as [Rei91] and [Vau87], we give a different semantics for the multisets, closer to the approach of [DHP91], modelling the place values. Instead of using a syntactic extension and taking the resulting models, we directly extend the models with a group structure (for the multiset union, difference and emptyset).

3.2.1 Multisets

In order to be able to model the multiset of values for each place we must add a multiset to each sort of the algebraic specification.

Given a specification $SPEC = (S, \leq, F, X, Ax)$, the extension $[SPEC]$ is done by adding respectively the new sorts $[s]$ and the following operations to S and F, for all s in S:

1) $\emptyset_{[s]} : \rightarrow [s]$	3) $+_{[s]} : [s]\ [s] \rightarrow [s]$	5) $\subseteq_{[s]}: [s]\ [s] \rightarrow \{\text{true}, \text{false}\}$
2) $\text{Single}_{[s]} : s \rightarrow [s]$	4) $-_{[s]} : [s]\ [s] \rightarrow [s]$	

If we consider the models of the specification *Alg(SPEC)*, all algebras A are extended into $[A]$ with the following multiset structure. Given a set E, the multiset $[E]$ is $\{f \mid f : E \rightarrow \mathbf{N}\}$ with the following operations:

1) $\forall e \in E,\ \emptyset_{[E]}(e) = 0$

2) $\forall d, e \in E,\ \text{Single}(d)_{[E]}(e) = \begin{cases} 1 & \text{if } e = d \\ 0 & \text{otherwise} \end{cases}$

3) $\forall e \in E,\ (\forall f, g \in [E]), (f +_{[E]} g)(e) = f(e) +_{\mathbf{N}} g(e)$

4) $\forall e \in E,\ (\forall f, g \in [E]), (f -_{[E]} g)(e) = f(e) -_{\mathbf{N}} g(e)$ if $f(e) \geq g(e)$ otherwise not defined

5) $\forall f, g \in [E],\ f \subseteq_{[E]} g = \begin{cases} \text{true} \Leftrightarrow \forall e \in E f(e) -_{\mathbf{N}} g(e) \geq 0 \\ \text{false otherwise} \end{cases}$

3.2.2 Algebraic net syntax

We give in this section the syntactic elements needed to define a class of algebraic nets. We consider in this part a universe U and the new disjoint subsets: the set of place names *PLACES*, the set of transition names *TRANSITIONS* and the set of *VAR* of variable names.

Given $P \subseteq PLACES$ an $(S^* \rightarrow S)$-set of places, an *S-indexed P-indexed* term t is a term satisfying the following properties: $\forall w \in S^*, \forall s \in S, \forall p \in P_{w \rightarrow s}, t_p \in (\mathrm{T}_{[SPEC],X})_{[s]}$. The set of all S-indexed P-indexed term is noted $\mathrm{T}_{P,[SPEC],X}$.

Definition 3.10 *Algebraic net specification*
Given $SPEC = (S, \leq, F, X, Ax)$ a specification, $P \subseteq PLACES$ an $(S^* \rightarrow S)$-set of place names, $T \subseteq TRANSITIONS$ a set of transition names, $X \subseteq VAR$ an S-set of variables. An *algebraic net* is a quadruplet $AP = (SPEC, P, T, X, BAx)$. Where BAx is the set of behavioural axiom, an axioms is a term $Cond \Rightarrow ev :: In \rightarrow Out$ defined as :

1) $Cond$ is $t_1 = t'_1 \wedge t_2 = t'_2 \wedge \ldots \wedge t_n = t'_n$ where $t_i, t'_i \in (T_{\Sigma,X})_{s_i}$

2) $ev \in T$

3) In and Out are S-indexed P-indexed terms

We also add an initial marking *INIT* to the definition of one algebraic net. An initial marking gives a unique starting point to the behaviour of the algebraic net. *INIT* is an S-indexed P-indexed term.

3.2.3 Semantics of algebraic nets

The semantics of the algebraic nets is based on transition systems over states called net marking and labelled by the transition names. The markings are functions from the set of places to the place values which are given by the models of the algebraic specification.

Given $SPEC = (S, \leq, F, X, Ax)$ a specification, $A \in Alg([SPEC])$ and given P a set of places, $V_{P,A} = \{m \mid m : P \rightarrow A\}$ where $\forall w \in S^*, \forall s \in S, \forall p \in P_{w \rightarrow s}, \forall m \in V_{P,A}\, m(p) \in A_{[s]}$

Given $SPEC = (S, \leq, F, X, Ax)$ a specification and $AP = (SPEC, P, T, X, BAx)$ an algebraic net a *Net-structure* is a pair (A, TS_A) where:

1) $A \in Alg([Spec])$

2) $TS_A \subseteq (V_{P,A} \times [T] \times V_{P,A})$

The transitions are labelled by multiset of label names, expressing the possible steps (multiset of transition names) that can be performed by the model.

Moreover we have to canonically extend the operations (already defined on markings) on multisets construction. Namely: sum, difference and inclusion.

1) $+_{P,A} : V_{P,A} \times V_{P,A} \rightarrow V_{P,A}$

2) $-_{P,A} : V_{P,A} \times V_{P,A} \rightarrow V_{P,A}$

3) $\subseteq_{P,A}: V_{P,A} \times V_{P,A} \rightarrow \{\text{true, false}\}$

For a given assignment σ the interpretation of terms μ_σ from $T_{\Sigma,X}$ into A is also extended for the interpretation on S-indexed P-indexed terms to $\mu_{P,\sigma}$ from $\mathrm{T}_{P,[SPEC],X}$ into $V_{P,A}$.

Definition 3.11 *Algebraic nets semantics*
Given a specification $SPEC = (S, \leq, F, X, Ax)$ and an algebraic net $AP = (SPEC, P, T, X, BAx)$, a Net-structure $N = (A, TS_A)$ is valid if the following holds

$$\forall m, m' \in V_{P,A} \forall e \in [T], m \xrightarrow{e} m' \in TS_A \Leftrightarrow$$
$$\begin{cases} \exists bax \in [BAx] \text{ and an assignment } \sigma : X \text{ into A} \\ e = \sum_{Cond \Rightarrow ev::In \rightarrow Out \in bax} ev \\ (\sum_{Cond \Rightarrow ev::In \rightarrow Out \in bax} \mu_{P,\sigma}(In)) \subseteq_{P,A} m \\ \forall (Cond \Rightarrow ev :: In \rightarrow Out \in bax) \ \langle A, \sigma \rangle \models Cond \\ m +_{P,A} (\sum_{Cond \Rightarrow ev::In \rightarrow Out \in bax} \mu_{P,\sigma}(Out)) = m' +_{P,A} (\sum_{Cond \Rightarrow ev::In \rightarrow Out \in bax} \mu_{P,\sigma}(In)) \end{cases}$$

From the previously defined model which do not include initial states we add a condition taking into account the definition of an initial state term description *INIT*. We select all transitions having a reachable origin from the initial marking and denote the model satisfaction by $AP, \mu_{P,\emptyset}(INIT) \models BAx$.

3.3 Hierarchical Specifications

In the previous sections we introduced the basic models: Algebraic Specification and Algebraic Petri Nets. In this section we will show how to enhance those models by adding structuring principles. We first introduce the general principle of hierarchical specification which is based on structuring operators. We introduce also the principles used for connecting the modules within a hierarchy. In the case of algebraic specification the hierarchy is naturally captured by the function calls organized into layers by the axiom definition. In Petri nets there is no natural way to have interconnections. We introduce for that purpose a synchronization mechanism including several operators such as the simultaneous synchronization, the sequential and the non-deterministic synchronization and an abstraction operator.

3.3.1 Structuring operators

We use here the union '$\amalg$' and enrichment '$\triangleleft$' operators already used in [Bid89b]. A specification is the composition of modules using such kind of operators. The union operator '$\amalg$' performs the union of two specifications while the enrichment operator '$\triangleleft$' adds a partial specification to a given specification. The empty specification allows to start the hierarchy.

3.3.2 Partial Specifications

A partial specification is a specification of algebraic nets which is defined over a larger signature than the one defined inside the module. A *partial specification* $\triangle SPEC = (S, \leq, F, X, \triangle Ax)$ and $\triangle AP = (SPEC, P, E, X, \triangle BAx)$ is similar to an algebraic net specification except that $\triangle Ax$ and $\triangle BAx$ are defined over all possible sorts in *SORTS* and functions in *FUNC*. Moreover, the events are differently defined. Instead of having a finite set of transition names, we introduce for each partial algebraic net a $SORTS^*$-indexed family $E = (E_w)_{w \in SORTS^*}$. The events are used to build an algebra over values of Σ-algebra with the following abstraction and synchronization operators: '$\emptyset$' for invisible events, '`&`' for simultaneousness, '`..`' for sequence, '+' for non-determinism and **`With`** for abstraction.

1) $\emptyset \in EVENTS(A)$

2) $\forall e, e' \in EVENTS(A), \quad e\&e' \in EVENTS(A),$

3) $\forall e, e' \in EVENTS(A), \quad e..e' \in EVENTS(A),$

4) $\forall e, e' \in EVENTS(A), \quad e{+}e' \in EVENTS(A)$,

5) $\forall (a_1, \ldots, a_n) \in A_{s_1} \times \ldots \times A_{s_n}, \forall e : s_1, \ldots, s_n \in E$,
$e(a_1, \ldots, a_n) \in EVENTS(A)$,

6) $\forall (a_1, \ldots, a_n) \in A_{s_1} \times \ldots \times A_{s_n}, \forall e' \in EVENTS(A), \forall e : s_1, \ldots, s_n \in E$,
$e(a_1, \ldots, a_n)$ **With** $e' \in EVENTS(A)$.

The terms appearing into the axioms of the partial algebraic nets are taken from $EVENTS(T_{\Sigma,X})$.The semantics of one partial specification is then, for the algebraic net part, a transition system , for a given algebra A labelled by values of the $EVENTS(A)$ algebra.

3.3.3 Well Structured Specification

Starting from a hierarchical specification, the corresponding specification is built from the hierarchy of modules by flattening the structure. The specification is simply obtained by recursive concatanation of the modules.

1) $Spec(\emptyset) = \emptyset$

2) $Spec(SPEC_1 \amalg SPEC_2) = Spec(SPEC_1) \cup Spec(SPEC_2)$

3) $Spec(SPEC_1 \lhd \triangle SPEC) = Spec(SPEC_1) \cup Spec(\triangle SPEC)$

It must be noted that a correct hierarchical specification must be a complete flattened specification for each of its sub-trees (in order to have a pure hierarchy of modules). It means that all axioms must be built over known signature and functions.

Moreover, it is necessary to add some conditions to a hierarchical specification, concerning the shared part of the tree of the hierarchy. In concrete specification it seems natural that part of the specification are built over the same module. Unfortunately, it is not possible to express such sharing using the proposed structuring operators. In this case it is necessary to have a copy of the common specification and to impose a syntactic and semantic constraint on the common parts.

3.4 Semantics of Hierarchical Specifications

The semantics of hierarchical algebraic nets is mainly based on the semantics of algebraic abstract data types and algebraic nets. Nevertheless, the structural operators impose some conservation properties between each level of the hierarchy. These properties are basically the hierarchical consistency and the sufficient completeness for the algebraic abstract data types, expressing the robustness of the hierarchical extension. In the context of algebraic nets the intended properties to satisfy are simpler to express due to the operational nature of the model. Regarding object-orientation, a clear property to be satisfied is the encapsulation. Encapsulation expresses that only the interface is visible to the above hierarchy and, consequently, the behaviour do not depend on any particular definition choice. In order to define the semantics of hierarchical algebraic nets, we extend some definitions of algebraic nets such as the markings, we introduce some semantics operators allowing to gather the semantics of each module of the specification. They are the behavioural synchronization closure *Sync*, and stabilization closure *Stab* for the algebraic nets and the model preserving extensions for the hierarchical algebraic specifications.

The partial semantics *PHAN-Sem* of the partial specifications set is built from the concatenation of the semantics of all partial specifications *AN-Sem* and further from the application of the Closure and Synchronization operators.

1) $PHAN\text{-}Sem(\emptyset) = \emptyset$

2) $PHAN\text{-}Sem(SPEC_1 \amalg SPEC_2) = PHAN\text{-}Sem(SPEC_1) \cup_{\mathrm{AN-structure}} PHAN\text{-}Sem(SPEC_2)$

3) $PHAN\text{-}Sem(SPEC_1 \lhd \triangle SPEC) = PHAN\text{-}Sem(SPEC_1) \cup_{\mathrm{AN-structure}} AN\text{-}Sem(\triangle SPEC)$

3.4.1 Marking Structures

The markings are defined over all possible places, $+_{P,A}$, $-_{P,A}$, $\subseteq_{P,A}$ are extended canonically to the operation on all possible places $+_{PLACES,A}$, $-_{PLACES,A}$, $\subseteq_{PLACES,A}$, considering that undefined image value of the multiset functions corresponds to empty set.

3.4.2 Behavioural Closure

This closure is used to compute all deductible behaviours from the semantics of the basic module, adding concurrent and sequential behaviour. The models, as in algebraic nets, are expressed using labelled transition system. They are called partial because the labels use the abstraction operator to express the needed synchronization. The synchronization closure will eliminate such synchronization by producing a complete model. In order to achieve that we introduce some basic operators on marking such as $m_1 \bowtie m_2$ and $m_1 \trianglelefteq_{HAN} m_2$. $m_1 \bowtie m_2$ means that two markings are equal on their common places. $m_1 \trianglelefteq_{HAN} m_2$ considers two marking and gives a marking with the value of the marking m_1 plus the value of the places of m_2 not in m_1.

$$\begin{aligned} Beh(str) \;&=\; str \cup \{m_1 \trianglelefteq_{HAN} m_2 \xrightarrow{e_1..e_2} m_2' \trianglelefteq_{HAN} m_1' \mid m_1' \bowtie m_2 \wedge \forall m_1 \xrightarrow{e_1} m_1', m_2 \xrightarrow{e_2} m_2' \in str\} \cup \\ &\quad\; \{m_1 + m_2 \xrightarrow{e_1 \& e_2} m_1' + m_2' \mid \forall m_1 \xrightarrow{e_1} m_1', m_2 \xrightarrow{e_2} m_2' \in str\} \end{aligned}$$

3.4.3 Synchronized Closure

In order to compute the full model of a set of algebraic nets, we have to solve the synchronization constraint among transitions. This means that we have to transform the $e\ With\ e'$ labels into simple labels.

1) $PreSync(str) = str \cup \{m_1 + m_2 \xrightarrow{e_1} m_2' + m_1' \mid \forall m_1 \xrightarrow{e_1 With\ e_2} m_1', m_2 \xrightarrow{e_2} m_2' \in str\}$

2) $Sync_0(str) = str$

3) $Sync_n(str) = PreSync(Beh(Sync_{n-1}(str)))$

$$Sync(str) = \lim_{n \to \infty} Sync_n(str)$$

Remarks: This construction exists because of the monotonicity of the operators.

3.4.4 Stabilization Closure

The behaviour of an algebraic net is based on two classes of events: the immediate events written $\emptyset$ and the named events. The immediate events must not appear in the semantics of the algebraic nets, because they model invisible reactions of the modelled system.

1) $PreStab(str) = str \cup \{m_1 \trianglelefteq_{HAN} m_2 \xrightarrow{e_1} m_2' \trianglelefteq_{HAN} m_1' \mid m_1' \bowtie m_2 \wedge \forall m_1 \xrightarrow{e_1} m_1', m_2 \xrightarrow{\emptyset} m_2' \in str\}$

2) $Stab_0(str) = str$

3) $Stab_n(str) = PreStab(Stab_{n-1}(str)))$

$$Stab(str) = \lim_{n \to \infty} Stab_n(str)$$

The stabilization is built for every hidden path of any length. In order to have a finite behaviour it is necessary that the fixpoint is reached for a bound k.

3.4.5 Semantics of hierarchical algebraic specification

The semantics of hierarchical specification is based on the classical notion of models finitely generated. In particular the basic modules are initial models in order to fix the hierarchy build over them to interesting hierarchical models, for instance, true $\neq$ false imposes indirectly some restrictions on the model of the sorts defining observers on boolean.

1) $HMod(Spec)$ is the category of hierarchical models

2) $\_|_{Spec}$ is the forgetful functor.

3) $Gen(Spec)$ is the category of finitely generated models

The hierarchical models are inductively defined over the structuring operators: $HMod(Spec) \subseteq Gen(Spec)$

$HMod(Spec \lhd \triangle Spec) =$

1) $Spec = \emptyset \Rightarrow \{I \in Gen(\triangle Pres) \mid I \text{ is the initial model}\}$

2) $Spec \neq \emptyset \Rightarrow \{A \in Gen(Spec \lhd \triangle Pres) \mid A|_{Spec} \in HMod(Spec)\}$

$HMod(Spec \cup Spec') = \{A \in Gen(Spec \cup Spec')|A|_{Spec} \in HMod(Spec) \text{ and } A|_{Spec'} \in HMod(Spec')\}$

3.4.6 Semantics of hierarchical algebraic nets

$HAN\text{-}Sem(Spec) = \{\langle A, TS_A\rangle \mid TS_A$ is a transition system and $A \in HMod(Spec)\}$ is the semantics of hierarchical nets expressed over the semantics of the algebraic specification part $A \in HMod(Spec)$ and the semantics of the algebraic nets expressed by the transition system TS_A. The selector $\_|_{HNet}$ and $\_|_{HMod}$ selecting one or another part of the semantic structure.

$HNet(Spec \lhd \triangle Spec) =$

1) $\triangle Spec =$ Algebraic module $\{\langle A, \emptyset\rangle A \in HMod(Spec \lhd \triangle Spec)\}$

2) $\triangle Spec =$ Net module

$\{\langle A, Stab(Sync(AN\text{-}Sem(\triangle Spec) \cup TS_A))\rangle \mid TS_A \in HAN\text{-}Sem(Spec)|_{HNet}, A \in HAN\text{-}Sem(Spec)|_{HMod}\}$

$HAN\text{-}Sem(Spec \amalg Spec') = Stab(Sync(HAN\text{-}Sem(Spec) \cup HAN\text{-}Sem(Spec')))$

3.4.7 Encapsulation

A system is encapsulated iff the hierarchy do not see the local state of the objects and their depending sub-objects. In the context of hierarchical specification, encapsulation is guaranteed if places are clearly encapsulated into each algebraic net composing the hierarchy. It means that the places used by the axioms must only be used by events of the same module. This condition can be syntactically expressed.

4 Semantics of CO-OPN/2 with HAP nets

A CO-OPN/2 specification is a HAP net specification with some additional notions concerning the Petri net part such as:

1) a set of type names, a type is a collection of methods (signature part) and a set of places (states) with axioms describing its behaviour,

2) a subtype relation among the type names,

3) an instanciation mecanism allowing to create objects,

4) the M method call for the object O is expressed by the pointed expression $O.M$

In this paper we avoid all explanative aspects which concern syntactic aspects such as visibility, renaming and the parametrization aspects borrowed from Pluss [Bid89a].

We give a first semantics of CO-OPN/2 in terms of translation of the CO-OPN/2 construct into HAPnets. The principle of the translation is mainly that classes are net modules, in which object states are represented by couples $\langle oid, value\rangle$ where the object identifier is a value *oid* of a reference sort and value represents the state of the place (it is distributed on the class places). Moreover, inheritance is described by delegation of the methods to its super-type. Object reference is described by ordered sorts.

4.1 Class reference

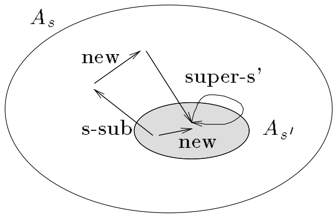
Each class name is described by a sort. The inheritance relation is exactly described by the order relation on that sort.

For a given pair of sorts s and s' describing the class references S and S' with a subtype relation $s' < s$ we introduce the following reference sort definition, which is defined as the subtype relation.

```
Module Spec Ref';
Signature
 Sorts s';
 Operations
 init     :    → s';
 new      : s' → s';
 super-s' : s' → s';
Body
 Axioms
 super-s' c' = c';
 Where
 c': s';
End Ref';
```

```
Module Spec Ref;
Signature
 Sorts s;
 Ordering s' < s;
 Operations
 s-sub    : s → s ;
 super-s' : s → s';
 new      : s → s ;
Body
 Axioms
 s-sub(s-sub(c)) = c;
 super-s'(s-sub(c)) = super(c);
 new(s-sub(c)) = s-sub(new(c));
 Where
 c: s;
End Ref;
```

The reference sorts are built using the `init` and `new` operations available in every supersort of the hierarchy. The `s-sub` gives the possibility to find a value of an higher sort and the `super-s` gives the possibility to transform a reference of any value of the hierarchy into a reference value of the `s` sort. In terms of model, the figure on the right expresses the carrier set relation and the operations.



4.2 Class States and Class Methods

The place of a given class *class* is $p : t$ corresponds to $p : class \times t$. The method of a given Class *class* is $m\_\_ \ldots \_ : t_1, t_2, \ldots, t_n$ corresponds to $\_m\_\_ \ldots \_ : class, t_1, t_2, \ldots, t_n$.

4.3 Class Axioms

The axioms for the class *class* is $Cond \Rightarrow ev :: In \rightarrow Out$ corresponds to the axioms where

1) ev is changed into ev' (each method call adds a parameter, which is the object identifier), $ev' = self\ ev$. Moreover the method calls appearing in the synchronization part are themselves modified using the pointed notation, $o.mexp$ is translated to $super(o).m$ if m is a method of the super type or unchanged if m is defined in the type (the process is recursive on the subtype relation).

2) In is changed into In' (each place term is extended to couple including the object identifier), $In = p_1\ exp_1, p_2\ exp_2, \ldots, p_n\ exp_n$ is translated into $In' = p_1\ self\ exp_1, p_2\ self\ exp_2, \ldots, p_n\ self\ exp_n$ where *self* is a variable of type class.

3) Out is changed into Out' (each place term is extended to couple including the object identifier) similarly as In.

4.4 Class instanciation and Delegation

The way used to build a new object from a class is to define a new *oid* for the object through the use of an operation on the sort describing the references of that class. This process has a side effect on the class because it is necessary to know a new *oid* from the previously defined *oid*. A possible way used in our approach is to include an order notion by the computation of the successor (*new*).

```
Module Class class';
Signature
 Type s';
 Methods
 basic;
Body
 Axioms
 basic;
End class;
```

```
Module Class class;
Signature
 Type s;
 Ordering s < s';
 Methods
 newobject;
Body
 Axioms
 newobject With c.create .. c.basic;
End class;
```

These two frames correspond to the following HAP nets (using a syntax closed to the CO-OPN/2 where the **Type** fields disappear and are replaced by sort definition used as reference.

```
Module Class class';
Signature
 Methods
 _ create: s';
 _ basic : s';
Body
 Places
 oid: → {s'};
 Axioms
 new c' create : oid{c'}
  → oid{new c'};
 c' basic;
 Where
 c':s';
End class;
```

```
Module Class class;
Signature
 Methods
 _ create    : s ;
 _ newobject : s ;
Body
 Places
 oid : → {s};
 Axioms
 (new s-sub c') create With
  c' create : oid{c} →
   oid{new s-sub c'};
 self newobject With
  c create .. super-s'(c) basic;
 Where
 self,c: s; c':s';
End class;
```

4.5 Behavioural subtyping

The syntactic relation between subtype is not sufficient to guarantee the principle of substitutability of a type by its subtype. A semantic condition (not detailed here) must be added. This condition is based on the behavioural equivalence between objects (a weak bisimulation) generalized to classes.

5 Conclusion

In this paper, we have shortly presented an extension of CO-OPN language called CO-OPN/2 where class and subtyping notions have been introduced. The semantics of CO-OPN/2 has been expressed by means of HAP nets in the same manner as [BG93] which is a complete formalism allowing description of modular systems.

Future works will be carried out in three directions:

- the description of the semantics of CO-OPN/2 will be improved, especially concerning the refinement aspects,
- tools supporting the language for the already developed environment SANDS will be built,
- CO-OPN/2 will be experimented on practical case studies.

Acknowledgements

We wish to thank M. Buffo, N. Guelfi and P. Racloz for their numerous and valuable comments on this study.

References

[AR93] E. Astesiano and G. Reggio. Algebraic specification of concurrency. In Michel Bidoit and Christine Choppy, editors, *Recent Trends in Data Type Specification, selected papers*, volume 655 of *LNCS*, pages 1–39. Springer-Verlag, 1993.

[BC93] Eugenio Battiston and Fiorella De Cindio. Class orientation and inheritance in modular algebraic nets. In *Proceedings International Conference on Systems, Man and Cybernetics*, volume 2, pages 717–723, Palais de L'Europe Hôtel Westminster, Le Touquet - France, October 1993. Ecole Centrale de Lille.

[BCM88] Eugenio Battiston, Fiorella De Cindio, and G. Mauri. OBJSA nets: a class of high level nets having objects as domains. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1988*, volume 340, pages 20–43. Springer-Verlag, 1988.

[BFR93] Didier Buchs, Jacques Flumet, and Pascal Racloz. SANDS: Structured algebraic net development system. In Buy Ugo, editor, *14th International Conference on Application and Theory of Petri Nets, Tool presentation abstracts*, pages 25–29. Chicago, USA, June 1993.

[BG91] Didier Buchs and Nicolas Guelfi. A concurrent object oriented Petri nets approach for system specification. In M. Silva, editor, *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Aahrus, Denmark, June 1991.

[BG93] Didier Buchs and Nicolas Guelfi. Formal development of actor programs using structured algebraic Petri nets. In Bode Arndt, Reeve Mike, and Wolf Gottfried, editors, *Int. Conference on parallel Architectures and Languages Europe PARLE'93*, volume 694, pages 353–366. ECRC, Munich Germany, Lecture Notes in Computer Science, Springer-Verlag, June 1993.

[Bid89a] M. Bidoit. *Pluss, une langage pour le developpement de specifications algebrique modulaires*. PhD thesis, Universite de Paris-Sud, Centre d'Orsay, 1989.

[Bid89b] M. Bidoit. The stratified loose approach: a generalization of initial and loose semantics. In *Proc. 1st International Conference on Algebraic Methodology and Software Technology*, Iowa City, USA, 1989.

[BP90] R. Bastide and P. Palanque. Petri nets with objects for the design, validation and prototyping of user-driven interfaces. In D. Diaper et al., editor, *Human-Computer Interaction: INTERACT'90*, pages 625–631, Amsterdam, North-Holland, 1990. Elsevier. Published as Proceedings of the IFIP TC 13 Third International Conference.

[Bre93] Ruth Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*, volume 562 of *LNCS*. Springer-Verlag, June 1993.

[DHP91] C. Dimitrovicci, U. Hummert, and L. Petrucci. Semantics, composition, and properties of algebraic high-level nets. In *Advances in Petri Nets*, volume 524 of *LNCS*, pages 93–117. Springer-Verlag, 1991.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 : Equations and Initial Semantics*, volume 6 of *EATC Monographs*. Springer-Verlag, 1985.

[FM90] J. Fiadero and T. Maibaum. Describing, structuring and implementing objects. In J. W. Bakker, W. P. Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX school proc.*, volume 489 of *LNCS*, Noordwijkerhout, The Netherlands, 1990.

[GM89] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions, and patial operations. Technical Report SRI-CSL-89-10, Computer Science Lab, SRI International, July 1989.

[Gue94] Nicolas Guelfi. *Les rseaux algbriques hirachiques: un formalisme de spcification structure pour le dveloppement de systmes concurrents*. PhD thesis, Universit Paris-Sud, LRI, September 1994. (to appear).

[JR91] K. Jensen and G. Rozenberg, editors. *High-Level Petri Nets*. Springer-Verlag, 1991.

[Lak94] C. A. Lakos. Object Petri nets, definition and relationship to coloured nets. R94-3, University of Tasmania, Dept. Comp. Science, Networking Research Group, April 1994.

[Reg93] G. Reggio. Event logic for specifying abstract dynamic data types. In *Recent Trends in Data Type Specification: selected papers*, volume 655 of *LNCS*, pages 292–309. Springer-Verlag, 1993.

[Rei85] Wolfgang Reisig. Petri nets. an introduction. In W. Brauer, G.Rozenberg, and A. Salomaa, editors, *EATCS Monographs on Theoretical Compute Science*, volume 4. Springer-Verlag, 1985.

[Rei91] Wolfgang Reisig. Petri nets and algebraic specifications. In *Theoretical Computer Science*, volume 80, pages 1–34. Elsevier, 1991.

[SB94] C. Sibertin-Blanc. Cooperative nets. In Robert Valette, editor, *Application and Theory of Petri Nets 1994*, volume 815 of *LNCS*, pages 471–490, 15th International Conference, Zaragoza, Spain, June 1994. Springer-Verlag.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86*, volume 21, 11 of *ACM SIGPLAN Notices*, pages 38–45, November 1986.

[Vau87] J. Vautherin. Parallel systems specification with colored Petri nets and algebraic specification. In Grzegorz Rozenberg, editor, *Advances in Petri Nets*, volume 266 of *LNCS*, pages 294–308. Springer-Verlag, 1987.