FORMAL DEVELOPMENT OF ACTOR PROGRAMS USING STRUCTURED ALGEBRAIC PETRI NETS

Didier BUCHS

CUI
University of GENEVA,
24, rue du Général Dufour
Tel : (41 22) 705 76 19
Fax : (41 22) 705 77 80

e_mail : BUCHS@CUI.UNIGE.CH

Nicolas GUELFI

***LRI**, URA 410 CNRS,*
Bât. 490, University of PARIS XI
91405 ORSAY FRANCE
Tel : (33 1) 69 41 66 32
Fax : (33 1) 69 41 65 86

e_mail : GUELFI@lri.lri.fr

ABSTRACT*

This paper provides with an actor semantics using a formalism called CO-OPN (Concurrent Object Oriented Petri Nets) which is modular and incorporates both concurrency and data structuring features. Actor languages are the main model of concurrent object oriented languages, but they are rarely well formalized. CO-OPN is a structured extension of Petri nets and algebraic abstract data types. A CO-OPN specification is composed of a set of objects (algebraic abstract data types and Petri nets), and of a synchronous communication mechanism. These synchronisation mechanism are defined in order to keep object state encapsulation, allowing to specify independantly each object. The translation of an actor program into a CO-OPN specification is done by building objects for actors. The adequacy of CO-OPN to describe the semantics of concurrent object oriented languages is shown by this formal semantic description of actor languages. Some particular implementations are suggested, improving the concurrency of executions of actor programs.

Keywords : Specification and verification, Formal program development methodologies, modular specification, semantics, actor languages semantics, high level Petri nets, algebraic abstract data types.

* This work has been supported by the "Fond National Suisse de la recherche Scientifique" and by the "ESPRIT" basic research action,"DEMON", n° 3148.

1. INTRODUCTION

Actor languages **[AGHA 86]** are a popular model used to introduce concurrency in object oriented programming. The actor model presents many features like encapsulation, asynchronous communication and dynamic creation of actor that are well suited to object programming. The semantics of concurrent object oriented languages are generally not well defined (by instance, Concurrent Smalltalk **[YOKO 86]**), certainly because such semantics must incorporate operational semantics for the concurrent aspects and algebras for the manipulated datas. Even if actor models are sufficiently powerful without predefined data structures, it is not realistic to define actors without basic data types like integer, Boolean, ... Without data structure, actor modelling would lead to a complex and non natural semantics where aspects related to true concurrency of the system modelled and aspects describing the data structures would be confused.

In this paper we propose a semantics of actors in which implementation choices are as delayed as possible. Data structure part of actors is described by means of algebraic specifications. The modular aspects of actors are modelled with specific structuring primitives. The semantics proposed in the paper can then express true concurrency of the actor system modelled. The specification formalism used is CO-OPN **[B&G 90]** for Concurrent Object Oriented Petri Nets, which provides both concurrency description and data description. Such a formalism puts minimal constraints on the concurrent aspects and on the data structure aspects. It allows us to analyze concurrency aspects without having chosen a particular implementation of data structures (a particular model of the algebraic specification **[GOGU 78]**). For example, the choice of a particular algebra (a model of the algebraic specification) is not needed when using the natural number specification to model an actor addresses.

The CO-OPN specification language makes use of algebraic abstract data types **[HERI 85]** and Petri nets in a structured way. A CO-OPN specification consists of a set of objects described by algebraic Petri nets **[VAUT 87] [REIS 91]** (a class of high level nets) that are connected through a synchronous communication mechanism integrating sequential and concurrent conditions over the communications. Some work has been done **[B&G 90]** on the semantic description of CO-OPN and the analysis of CO-OPN specification such as object equivalences, object replacement and change of algebraic abstract data type models. These results as well as the semantics of CO-OPN cannot be fully detailed here but some basic ideas are presented, especially on the semantics of CO-OPN and on the analysis of the concurrency in an actor system. Concerning the types, classes and inheritance we do not present in this paper all the corresponding CO-OPN features but we try to give an idea of how treating these notions in a formal way.

The aim of this paper is to use a simple model of actors, in which all important concepts of actors can be found, and to model it by CO-OPN. We first discuss related work and formalism, then we introduce informally CO-OPN, and we present the translation of actor programs into the CO-OPN formalism on an example and discuss some particularities of our translation, especially on the concurrency aspects. Beyond this case study, we show the software engeering interest of CO-OPN in the modelling and development **[B&F&R 92]** of concurrent programs and in the complex specification of programming languages or computer programs.

1.1. Work on semantic description of Actors

We present, here, related work on the semantic description of actor languages and the formalism used.
The actor model has first been introduced in **[HEWI 73]**. This model has been the subject of many researches. Formal work can be found in **[CLIN 81]**, **[AGHA 86]** and **[JANS 89]**.

Let us present quite recent semantic approaches near CCS and net theory.

- In **[HOPK 89]** a theoretical model consisting of an extension of CCS including dynamicity (DCCS) is studied for the description of actor programs.
- Yield another variant **[NIER 90]**: CCS augmented with a pattern mechanism in order to be more expressive in the description of the data structures. The semantics of this language is an interleaving semantics but no real description of data structures is given.
- Another model called POTS (Parallel Object based Transition System) is introduced in **[ENGE 90]**. It impose an additional structure on the place of a Predicate/Transition net. Objects in a given state are represented by places and events by transitions. The structure of the objects is defined by three functions mapping places to the represented objects , their states (unborn, alive, or dead) and their acquaintances.
- A modelling of an actor program by one colored net is done in **[V&S 90]** without data types and modularity.

Data structure and modularity of actors are not both expressed by these semantics. Our aim is to show how our formalism, CO-OPN fulfil these two requirements. As the theory of a complete object oriented model of actors is not yet done, we do not consider such actors. Nevertheless we give an idea of the CO-OPN approach to these concepts.

1.2. Basic formalism

Some formalism related to CO-OPN are described in this paragraph. These formalisms allow to express two concepts: the concurrency specification and the data structure specification. Good candidates to formalize are, Petri Nets **[REIS 85]**, CCS **[MILN 89]** or CSP **[HOAR 85]**. On the other hand, data structure specification may be achieved through the use of algebraic abstract data type specification, which is well known for its good mathematical properties. While the study of behavioural properties is simplified with the notion of compositionality of CCS, the graphical representation of Petri nets provides a powerful tool for software conception.

Formalisms integrating both features are Lotos **[LOGR 88]** (CCS and algebraic abstract data types), Algebraic Petri Nets **[VAUT 87 & BUCH 88]**, Segras **[KRAM 87]**, OBJSA **[BATT 89]** (Petri Nets and algebraic abstract data types) and our one called Concurrent Object Oriented Petri Nets (CO-OPN) **[BUCH 90]** (Petri Nets, algebraic abstract data types and structuring).

The main advantages of CO-OPN with respect to the other formalisms are the structuring by objects (as in **[FIAD 90]** with deontic logic) and the respect of true concurrency. Moreover some interesting results on object equivalence are available, allowing to replace objects in a specification without change of the object behaviour. We illustrate in this paper the power of the CO-OPN formalism by the construction of a semantics for actor languages.

In this paper the full syntactic and semantic description of our approach is summarized because of the lack of space, we present our ideas by means of examples. In **[B&G 90]** a complete description of the semantics of CO-OPN is given by means of a concurrent distributed transition systems and algebras corresponding to an algebraic specification. In **[GUEL 90]** a complete and formal description of the semantics of actors is given.

2. CO-OPN

In this paragraph we briefly introduce the CO-OPN formalism starting with basic concepts : algebraic abstract data types, Petri nets and algebraic Petri nets. A CO-OPN specification is composed of objects described by algebraic Petri nets and a synchronization mechanism linking these nets together. As said before, these synchronisation mechanism are defined in order to keep object state encapsulation, allowing us to specify independantly each object. Some examples for the use of CO-OPN can be found in **[B&G 91B]** for a general view of system specification with CO-OPN, in **[B&G 91C]** the open distributed systems are studied and in **[B&G&M 91]** a CO-OPN semantics of OCCAM-2 is given.

2.1. Algebraic abstract data types

Since the algebraic specification approach is well known (see for instance **[E&M 85]**), only the key points are summarized here. All of our specification examples are written using the PLUSS language developed at LRI **[BIDO 87&89]**. The approach consists in describing a class of objects, usually called the sorts, by defining the set of functions which operate on that class. These functions are described syntactically by their names and domains. Their semantic content is defined by a set of axioms expressed by conditional equations on terms. The functions can be grouped into categories called generator functions (**"generated by"** part) and operators (**"operations"** part). The set of axioms (**"axioms"** part) must be finite, and the axioms only can contain operations (functions) on the defined sort or on other previously defined sorts (**"use"** part), universally quantified variables (**"where"** part), and conditional tests.

Such specifications are semantically described by algebras. One algebra is assigned with each sort of the specification. An algebra is a carrier set provided with a set of operations corresponding to the operations of the specification. The class of models of a specification, Alg(Spec), contains the algebras satisfying the axioms of the specification Spec. The class of models satisfying the axioms and some particular constraints of the specification language PLUSS **[BIDO 89],** is denoted Mod(Spec) and is a subclass of Alg(Spec).

The set of natural numbers, also specified in the PLUSS language is classically defined with the generators 'zero' and 'succ' and the operations 'add', 'dec' (for decrementation), '>', 'parity', 'one' and 'two'. A specification of Fifo structures is also classical and defined as a parametrized module with the generators 'empty' and 'cat' and the operations 'is-empty', 'del' and 'next' (figure 1). The formal parameter of the module Fifo is described in figure 2 as FifoNat which specifies the files Fifo of natural numbers. An instantiation of this specification is proposed in figure 3. The instantiation needs to define the equivalent sorts and operations between the formal parameter 'Elem' and the concrete parameter 'Nat ' by a signature morphism 'morph'.

```
spec : Fifo(Elem)
sort : fifo ;
use : Bool ;
generated by :    empty : → fifo ;
                  cat : elem * fifo → bool ;
operations :      is-empty : fifo → bool ;
                  del : fifo → fifo ;
                  next : fifo → elem ;
axioms : …… ;
end ;
```

figure 1

```
par : Elem;
sort : elem;
end ;
```

figure 2

```
spec : FifoNat as:
Fifo(Elem => Nat by morph);
where : morph: elem => nat;
renaming : fifo into fifonat ;
end ;
```

figure 3

2.2. Calculating the values of a model of the algebraic specification

If a specification is sufficiently described, terms can be evaluated by the use of rewriting techniques **[D&J 90]**. Each closed term (term without variables) can be rewritten into its normal form. In PLUSS the normal forms are expressed only by means of generators (the models of PLUSS are called finitely generated w.r.t. the generators).

Example: 'add(succ(succ(zero)),succ(zero))' is rewritten in 'succ(succ(succ(zero)))', which is equivalent to 3 in the initial model, using the rewrite rules deduced from the axioms of the specification of natural numbers. The choice of one set of rewrite rules implies the choice of a particular algebra (usually the initial algebras of the class of models). In our example, the initial model of 'nat' is N={0,1,2,3,4,...}.

2.3. Petri nets

The Petri nets, PN=<P,T,In,Out,M0>, are generally used to model the dynamic behaviour of discrete systems **[REIS 85]**. They are very well suited to represent systems that consist of several processes evolving in concurrency. The relation between the model and the system is easily established using the *places* (P) representing the states of the system and the *transitions* (T) for the actions that can occur in this system. Oriented links between places and transitions indicate preconditions (In) and post-conditions (Out) of an action. Petri nets have a nice graphical representation : circles represent the places, boxes represent the transitions and oriented arcs link nodes of different kinds. A particular state of the net is defined by the notion of *marking* where each place contains *tokens*. The particular marking M0 corresponds to the initial state of the system. The Petri net has a model of execution, namely the evolution of the *tokens*. This provides a description of the dynamic behaviour of the system.

2.4. Algebraic Petri nets

Algebraic Petri nets APN=<Spec,P,T,Cond,In,Out,M0> present two levels of specification:

- the algebraic abstract data types (Spec) describing the data structures,
- the net providing the description of the behaviour of the control structure.

The resulting net is therefore a kind of coloured Petri net **[JENS 81]**. The expressiveness of Petri nets is improved by associating instances of algebraic abstract data types to the places instead of anonymous tokens **[VAUT 87]**. A place contains values of a single sort. In fact, a place is a multi-set i.e. set with multiple occurrences of the same allowed values of a sort. To enhance the semantic power of an APN, a predicate is

attached to each transition (Cond). A transition is fired only if all input places contain the values defined in its input arcs and if the predicate is verified. Once a transition occurs, the values are consumed by the transition and removed from their places and put in variables. Some of them are used to compute the output values of the transition according to the term attached to its output places. The semantics of Algebraic Petri nets **[VAUT 87]** is expressed by a translation into colored Petri nets. For our purpose the semantics is directly described by means of concurrent transition systems **[B&G 91A]**, the semantics of APN is for a given algebra $A \in MOD(Spec)$ the concurrent transition system $CTS_A(APN)$ which is a transition system with concurrent firing of actions.

About the notation: [] indicates the empty multi-set; and [succ(zero),succ(succ(zero))] indicates syntactically a multi-set containing the numbers 1 and 2. The corresponding value in the algebra is: [1,2].

Let us present a very simple example: A set of natural numbers is given and we want to produce the sum of these natural numbers, knowing the size of the set. Modelling this problem with an APN (see figure 4), 4 places and 2 transitions are defined:

nb :	the set of natural numbers (nat)
counter:	the number of remaining natural numbers to be summed
total:	the actual sum
out:	the final result
sum:	sums the natural numbers one by one
end:	tests for final result

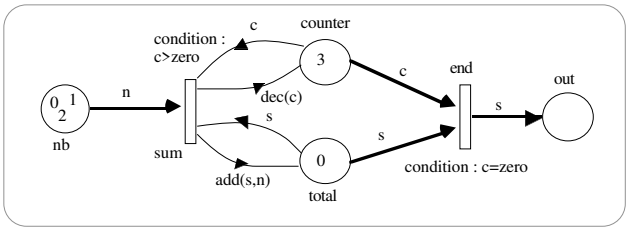

figure 4

The textual form of an APN (figure 5) is composed by seven fields separated by keywords:
- **apn :** the name of the algebraic net
- **use**: list of specifications used.
- **places**: enumeration of all couples formed by a place name and its type.
- **transitions**: list of the transitions names.
- **var**: enumeration of all couples formed by a variable name and its sort.
- **axioms**: description of the input place(s), output place(s) for the flot relation and predicate for each transition.
- **init_marking**: initial marking in term of multisets of values contained in each place.

```
apn:          Summation;
use:          Nat;
places:       counter,total,nb,out:nat;
transitions:  sum,end;
var:          c,n,s:nat;
axioms:
sum : >(c,zero) ⇒                                   (*the predicate is written before the arrow*)
      counter([c]) , nb([n]) , total([s])           (* input places with variables *)
      → total([add(s,n)]) , counter([dec(c)]);      (* output places with terms *)
end: c = zero ⇒ counter([c]) , total([s]) → out([s]);
init_marking: nb([succ(zero),succ(succ(zero)),zero]) , total([zero]) , counter([succ(succ(succ(zero)))]);
end;
```

figure 5

2.5. The transition system of an APN specification.

We give below the idea to built the semantics of an APN specification. For this the initial algebra model of an algebraic specification can be used to describe the semantics of the algebraic abstract data types. The APN semantics is then given using the definition of a concurrent transition system based on a particular algebra model of the algebraic specification. The states of the nets are built by extending this algebra into multisets which allow to concurrently fire one transitions (auto-concurrency).

We give an example of the computation of a transition of the concurrent transition system in the initial algebra (of natural numbers specification = N).
The initial marking is m_0 = nb([1,2,0]), total([0]), counter([3]), out([]), with this marking the net transition 'sum' can be fired with the variable instantiations {c=3, n=1, s=0}. After firing of 'sum', the marking is m_1 = nb([2,0]),total([1]),counter([2]),out([]). $m_0 \xrightarrow{[sum\{c=3, n=1, s=0\}]} m_1$ denotes this step of the transition system.

Remark: As APN have the computational power of Turing machines, computation of Petri nets classical properties is only possible if the construction of a finite graph covering the reachability graph is possible **[BUCH 88]**.

2.6. The CO-OPN formalism

2.6.1. Objects

A CO-OPN specification $SPEC_O$=<Spec,O,T_{ext},T_{int},P,AX> is first composed of a set of object O. Each object is an algebraic Petri net with two particular kinds of transitions: the first one are 'external' transitions called 'methods' (in the set T_{ext}), the second one are the 'internal' transitions (in T_{int}). The methods are parametrized transition and they give the only way to modify object states (the content in the places of the net) from the outside of an object. Only methods corresponding to events, reflect the object behaviour in the semantics. The internal transitions are non-parametrized transitions used to model the evolution of an object independently from the outside. The object states are the states of the places P of the objects, each place state is a value of the multisets built over the algebras of the place sorts. The behaviour of the net is described in the axioms. An axiom ax=<t,Cond,In,Out,SE> corresponds to the flow relation of the net transition 't' extended with the notion of synchronization SE defined in the following paragraph.

Example of an object:
We would like to describe an object called 'Buffer' (figure 6 & 7 for the textual and graphical forms) which is a fifo manager built over the 'fifonat' sort seen before. Two methods compose this object, 'get' and 'put', each method having as parameter the sort 'nat' . For each transition (internal or method) we give an axiom with the following syntax : *"transition_name_with_its_parameters : boolean_condition ⇒ input_relation → output_relation"*

```
object :      Buffer ;
use :         Nat,FifoNat,Bool;
methods :     get,put : nat;
places :      buffer : fifonat;
var :         b:fifonat; m:nat;
axioms :
        get(next(b)) : not(is-empty(b))
                ⇒ buffer ([b]) → buffer([del(b)]);
        put(m) : buffer ([b]) → buffer ([cat(b,m)]) ;
init_marking:      buffer([empty]);
end;
```

Figure 6

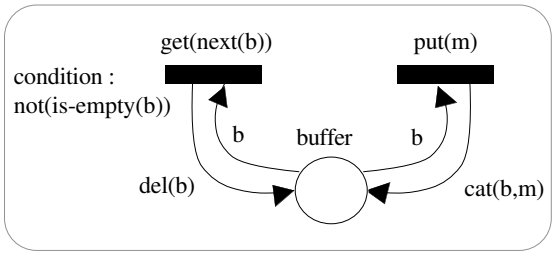

Figure 7

2.6.2. Synchronizations between objects

To construct a system from a set objects, we have to link them in a specially designed way. Instead of using classical message passing mechanism, the links between objects are defined as a synchronization expression "SE" composed with the binary operators 'sim' (for simultaneous), 'seq' (for sequence), the unary operator 'id' and the 'empty' synchronization.

For example, the construction 'a with sim(b,c)' where 'a','b','c' are methods, means that the method 'a' can be fired if the methods 'b' and 'c' can be fired simultaneously. In the same way 'a with seq(b,c)' means that the method 'a' can be fired if the methods 'b' and 'c' can be fired in sequence. The synchronization 'a with id(b)' means that the method 'a' can be fired if the methods 'b' can be fired (abbreviated to 'a with b'). Finally, the empty synchronization do not need any synchronization condition to fire a transition.

The simultaneous condition makes sense because the semantic of CO-OPN for a particular algebra A ∈ MOD(Spec) is given by means of concurrent transition systems $DTS_A(SPEC_O)$ expressing concurrency of actions. Moreover, to define the semantics of the object system it is necessary that the synchronization links between objects constitute a partial order (no cycles as in figure 8).

To show a CO-OPN object system we add to the previous object "buffer", two objects "consumer" and "producer". We first give their syntactic definition and then we build in figure 7 the whole object system with the synchronization expressions represented by symbolic links (stripped arrows). The behaviour of this system is classical. The relation between the objects is done by method calls. Once built a CO-OPN system we can give its semantics as described in the next section.

object : Producer ;
use : Nat,Buffer,Bool;
methods : send : nat;
places : counter : nat;
var : m,n:nat;
axioms :
send(m) **with** put(m) :
counter ([n]) → counter([add(succ(zero) , n)]) ;
init_marking: counter([zero]);
end;

object : Consumer ;
use : Nat,Buffer;
methods : receive;
var : m:nat;
axioms :
receive **with** get(m);
init_marking:
end;

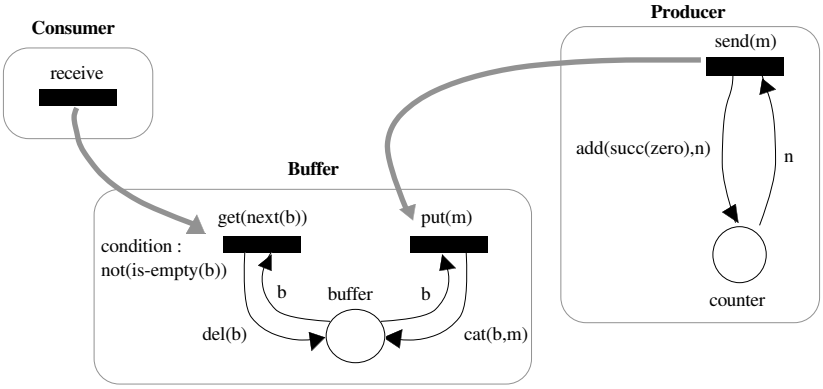

figure 8

2.7. The semantics of a CO-OPN specification

We compute the semantics of a CO-OPN specification (described in **[B&G 91A** by means of inference rules) in the same way as it was presented for the algebraic Petri Nets. The object system states are the states of all objects, the labels of the transition are multi-sets of concurrent actions, with one multi-set for each object. Then the semantics of an object system $SPEC_O$ is defined by the concurrent distributed transition system $DTS_A(SPEC_O)$ where A is a model of the algebraic specification.

Example : An algebraic model of the sort 'fifonat' can be composed, for example, with elements like this:
The value <> corresponds to evaluation of constant 'empty'.
The value <3,1> corresponds to evaluation of the term cat(cat(empty,succ(zero)),succ(succ(succ(zero)))).

The initial marking of the object systems is : m_0 = (object 'Producer':counter([0]), object 'Buffer' : buffer([<>])). (we do not mention the marking of the object "consumer" because it has no places) :
If we fire the method 'send(2)' then the marking is: m_1=('Producer' : counter([1]) , 'Buffer' : buffer([<2>])). And if we fire the method 'receive' from the marking m_1 then the resulting marking is: m_2=('Producer':counter([1]),'Buffer':buffer([<>]))> The step sequence of our DTS is :

$$m_0 \xrightarrow{<[send(2)],[],[]>} m_1 \xrightarrow{<[],[],[receive]>} m_2$$

To illustrate the semantics of the internal transitions a new example is constructed by changing '**methods** : receive' in the object 'Consumer' by '**internal** : receive'. This does not change this example because receive is not called by an other object, thus it doesn't need to be a method.
If the method 'send(2)' is fired from the initial marking m_0, the resulting marking is : m_3=('Producer':counter([1]),'Buffer':buffer([<>])). In the semantics of CO-OPN we force the internal transition 'receive' to be fired immediately after the firing of he method 'send(2)', so we write :

$m_0 \xrightarrow{<[send(2)],[],[]>} m_3$.

2.8. Parametrized CO-OPN objects.

In order to introduce generic definition of objects we define a notion of parametrized object. A parametrized object is an object that use special objects called parameter objects and special algebraic abstract data types called parameter specification. We give in the following an example of such object. It is the producer of figure 10 that use a parameter object (figure 9) to describe the possible communication mechanism. The parameter objects describe only the external view of an object (its methods).

```
param object :   ParamBuffer ;
use :            Nat;
methods :        getdata, putdata : nat ;
end;
```

Figure 9

```
gen object:  GenProducer(ParamBuffer) ;
use :              Nat;
methods :    send : nat;
places :     counter : nat;
var :        m,n: nat;
axioms :
send(m) with  putdata(m) :
        counter([n]) → counter([add(succ(zero),n)]) ;
init_marking: counter([zero]);
end;
```

Figure 10

In this ***figure 11*** we give a possible instantiation of this object can be:

```
object :   Producer as GenProducer(ParamBuffer => Buffer);
where :    getdata => get;
           putdata => put;
end;
```

Which produce the same meaning as the object Producer of figure 11. In addition it is possible to give an other parameter to the object for describing various data structure transmitted. The semantics of parametrized objects is noted $DTS_A(SPEC_O,Param)$ where Param is the specification of the parameters used. The semantics of abstract parameters $DTS_A(Param)$ is the semantics of an object without any places nor axioms. This gives a short idea of what could be the object oriented notions of class and inheritence in CO-OPN (for more details see **[B&G 91C]**).

2.9. Properties of CO-OPN formalism.

Some properties are formalized within the CO-OPN framework like object systems equivalence, object replacement and algebra replacement.

- The notion of equivalence **[POME 86]** between the semantics of object systems is defined for a fixed algebra A by the notion of concurrent bisimulation of two CO-OPN specifications $DTS_A(SPEC_O) \overset{\leftrightarrow}{\underset{c}{}} DTS_A(SPEC_{O'})$.
- One object can be replaced in an object system by an other one which is equivalent, producing a new object system bisimilar to the initial one.
- If two different algebras $A_1, A_2 \in MOD(Spec)$ observationally equivalent $A_1 \sim_{obs} A_2$ (see **[S&T 85]**) are used in the same object system then the two different object systems are equivalent by bisimulation under the assumption that the specification of the object system satisfy some syntactic conditions (free flow **[B&G 91A]**).

These properties allow us to refine an abstract CO-OPN specification into a more concrete one by progressively replacing abstract object and specification by more concrete one (see **[B&G 91B 91C]** for the theory and **[B&G 91B 91C]** for examples of application). These refinements notions are the main point of CO-OPN w.r.t specifications formalisms. They allow, considering the data and the system in a complete abstract way, to refine it while studying the properties kept at each step of the refinement.

3. ACTORS MODELLING USING CO-OPN

3.1. Introduction

In this section our method for the modelling of actor programs is described. First the general model for actors is introduced. Then the basic concepts of our method are presented. Finally an example including all the main features of actor languages is given and the corresponding CO-OPN specification is built with respect to the method described. A formal and complete description of our method is given in **[GUEL 90].**

3.2. The general model of actors

Actors languages are parallel languages. Then it is possible to create more actors to increase the parallelism of an actors system. The general idea is to split information and control as much as possible between actors. In an actors system the dynamic creation of actors and links between actors are allowed, communication is done by an asynchronous point to point message passing. To summarise, the actor model is a distributed model where knowledge and behaviour are distributed between actors which concurrently compute the tasks.

An actor has knowledge and rules to treat this knowledge:
- Local data (acquointances) which parametrize the actor behaviour and a parameter list of specific method datas as messages. Messages are stored in buffers.
- A behaviour, defining what the actor does during its life each time when messages arrive. More precisely an actor has a unique script activating the corresponding behaviour for each message.

In the general case two kinds of actors exist. If an actor is unserialized, a message treatment do not lead to a change of state. Then replacement behaviours with identical acquointances can be specified to treat concurrently messages in the buffer. Serial actors, where the treatment of messages leads always to a change of state. Then the replacement machine for the next message is created only after the change of state.

In the figure 12, an actor buffer is shown with its treatment machine. This machine treats a particular message of the buffer with respect to the order of arrival. When the machine A_n treats the n^{th} message, messages can be sent to other actors if knowing their address, new actors can be created and an other machine (A_{n+1}) must be created to treat the $(n+1)^{th}$ message. Before and after creating its replacement machine operations can be processed.

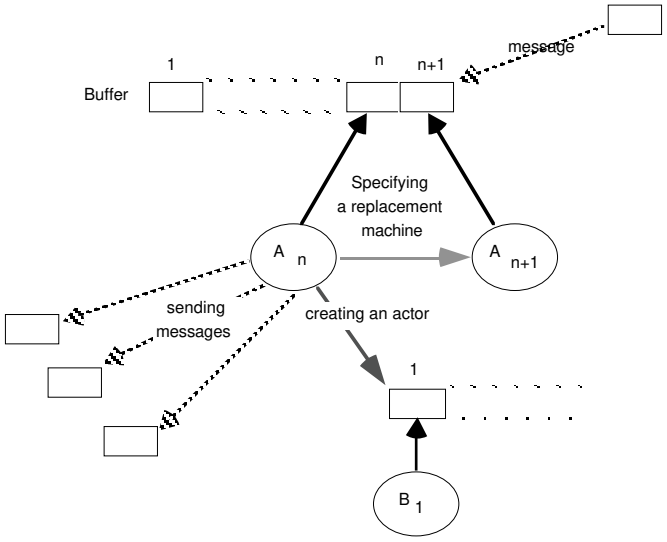

Figure 12

To integrate all these features, actor grammars must include three basic primitives : the 'send' command for message passing, the 'new' command for the creation of actors and the 'become' command for the specification of the actor replacement .

3.3. General concepts of the construction

The three important characteristics of actor languages are the concurrency aspects which are modelled by Petri nets, the data structures modelled using algebraic specification in actors languages and modularity modelled using CO-OPN objects structuring features. In order to model actor programs, a grammar must be given. A minimal abstract grammar of actor language is presented (the concrete one, used to write our program example, can be found in **[GUEL 90]**).

rule 1) <Prg> ::= <Actor>*
rule 2) <Actor> ::= ActorName **(** <Parameter>* **) [** <Method>* **]**
rule 3) <Parameter> ::= Ident **:** Type
rule 4) <Methode> ::= MethodName **(** <Parameter>* **) (** <Expression>* **)**
rule 5) <Expression> ::= **Send** MethodName*(*<*Term*>**)* **to** Ident
rule 6) | **Become** <ActorTerm>
rule 7) | **If** <BoolTerm> **then** <Expression>* **fi**
rule 8) | **If** <BoolTerm> **then** <Expression>* **else** <Expression>* **fi**
rule 9) | <Affectation>
rule 10) <Affectation> ::= **Let** Ident **:=** *Term*
rule 11) | **Let** Ident **:= New** ActorName*(*<*Term*>**)***

Once written an actor program a set of actors must be declared. These actors are the objects of our CO-OPN system. The construction of the CO-OPN system of an actor program is twofold :

1) the construction of the algebraic specifications in the PLUSS language
2) the construction of the object system in the CO-OPN language

3.3.1. Construction of the algebraic specification

The specification of the algebraic abstract data type is required to enable the formal use of data structures in actor programs. These structures are declared in the programs using the rule 3. Then the terms of these sorts can be used in the actor program (*Term* in the rules). On the other hand the algebraic specifications for the data structures defined for our construction (actors and buffers) are needed. An algebraic specification named #ActorNameTerm must be defined for : the actor structures ('act#ActorName') for the state of each actor and the message construction ('#ActorNameMethod'). An algebraic specification of the buffer structure must be added to define the sort 'buffer#ActorName' and the sort 'methodList#ActorName'. The principle for the construction of the algebraic specifications is to parse the actor programs and to build the corresponding algebraic specification needed (Remark : We use the '#' notation to indicate textual variables)

3.3.2. Construction of the CO-OPN system

Given an actor program, we construct the algebraic specification as explained before. Then the construction of the corresponding CO-OPN system can be undertaken; it is composed of a set of objects defining each actor class.

An actor class description can be divided into two parts : the first one defines an object for each method of the actor class and an object for the interface of the actor class. An interface for a class of actor is an object which synchronize all the evolutions of these actors. As a consequence, this interface contains transitions for the creation of a new actor of this class, for the reception of messages, for the treatment of messages and for the actor replacement. A place containing the buffers of each instanciated actor of this class and a place representing the instantiation of the actors of the same class are added to the interface object.
The general principle of these interfaces is that none of their transitions have synchronization expressions. This means that the synchronizations is induced by the method objects. For example, if in a method 'j' of an actor 'i' the code is " send a message k to an actor l ", the object corresponding to the method 'j' have to be synchronized with the interface of the actor l. (more precision on this point will be given below.)

When the CO-OPN objects specifying an actor 'Actor$_i$' are built, all instances of such actor have their own behaviour simulated by the same group of CO-OPN objects. For example, the places of the set of objects representing a class of actor 'Actor$_i$' contain all actors of type 'Actor$_i$' created in the whole system evolution. All instances are distinguished by their own address, memorized in the tokens. Note that a CO-OPN object simulates concurrently the behaviour of all actors of same class.

Example: Given an actor program 'P' as a list of actor declaration, all actors in 'P' have the following syntactic representation :

BegActor #Actor $_i$ (…… parameter list ……)
• • •
;
[#Method $_j$ (…… parameter list ……)
• • •
];
• • •
EndActor;

For each actor "Actor$_i$" the following objects are built (figure 13):

- an object for each method 'Method$_k$' to model the behaviour of this method.
- an object 'IActor$_i$' representing the interface of the 'Actor$_i$'.

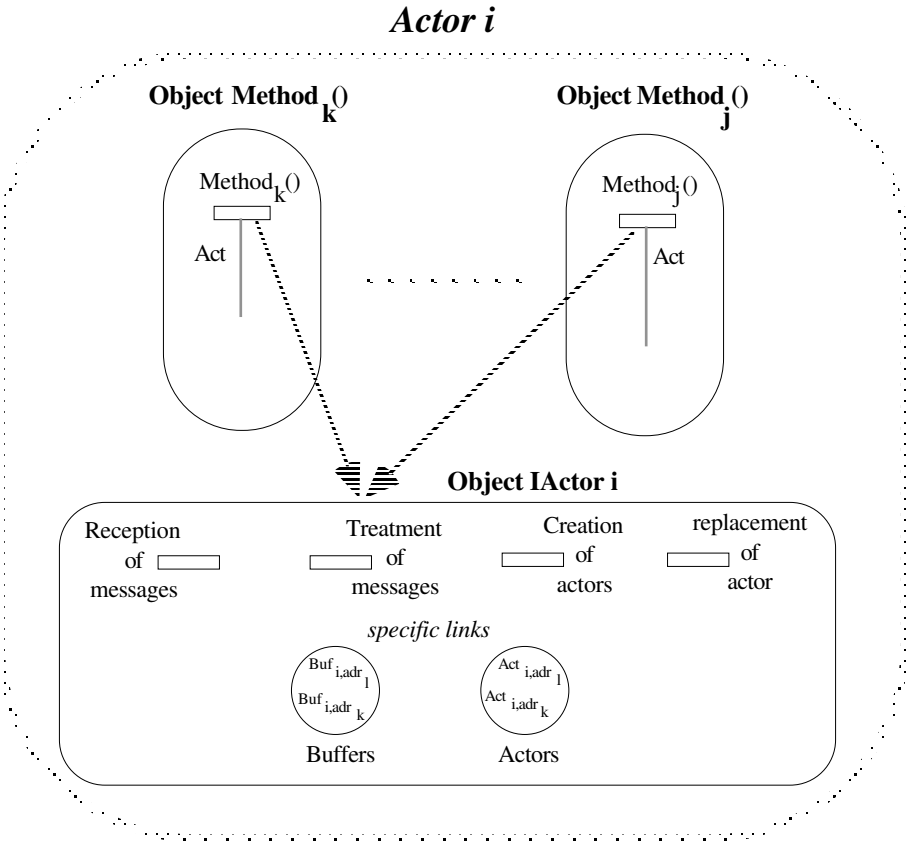

Figure 13

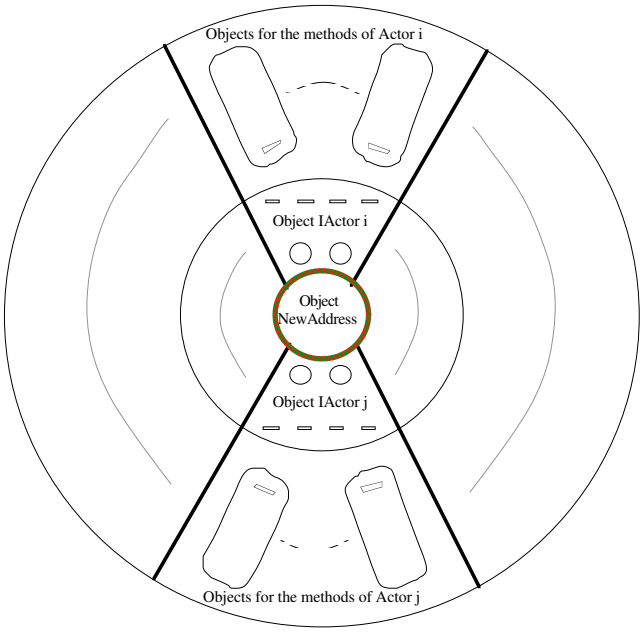

Figure 14

For an actor program with 'q' actors a CO-OPN specification is built with the structure represented in figure 14. In this figure the first layer corresponds to the interfaces object of the actors. In the second layer all the method objects of the actors are defined. The central object 'NewAddress' (figure 15 & 16 for the CO-OPN definition) is the address manager (for the creation of new address). The synchronizations between actors are always from a method object to an interface object (this point is described more precisely below) or to the 'NewAddress' object above.

```
spec Address as Nat
renaming nat into address
end;
```

Figure 15

```
object :        NewAddress ;
use :   Address;
methods : get_new_address:address ;
places :        next_address: address;
var :   m:address;
axioms :
get_new_address(m):
next_address([m]) → next_address([succ(m)])
init_marking: next_address([zero]);
end;
```

Figure 16

The method object

In this paragraph we describe the construction of a method object. When a method '$Method_j$' is declared in a actor program, the corresponding object is built. This object is constructed first by a transition named '$Method_j$ ()' linked to a place with an arc labelled by the actor variable representing the actor executing the method. This place is linked with the same kind of arc to a transition 'END' representing the end of the execution of the method '$Method_j$' (figure 17(a)). Then for each instruction of the actor program of this method, a transition simulating the action is added (figure 17(b)) and specific links are established with corresponding labels.
Six kind of transitions, one for each kind of expression, are allowed by our grammar. Each one having a precise CO-OPN specification and are formally defined in the associated axiom. They are described by our example in the next section.

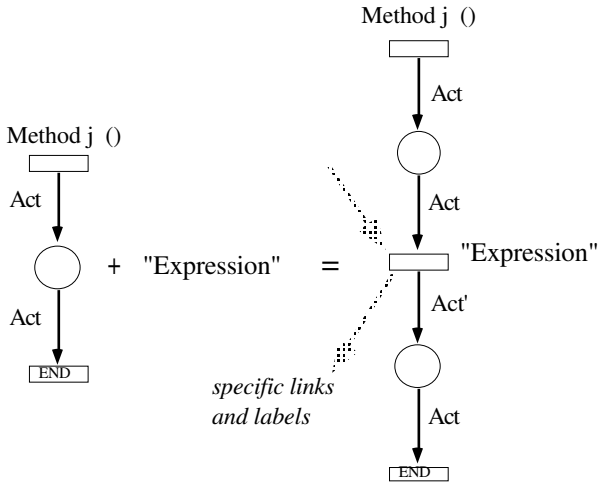

Figure 17 *(a)* *(b)*

The interface object

The interface objects have the same design for each actor. We now give a complete description of such an object, represented in the figure 18. We define it formally using the CO-OPN language :

This object has four transitions and two places:

- The transition 'receive(M,adr(B))' : This transition is called to execute the reception of a message 'M' in the buffer 'B' of an actor of type actor$_i$ with address 'adr(B)'.
- The transition 'Treat(next(B),Act)' : If this transition is in an interface object of an actor 'Actor$_i$' then it is synchronized with the first transition of each method object 'Method$_j$' of the actor 'Actor$_i$' modelling the message treatment.
- The transition 'create(Act,Adr)' : When an expression of an actor program corresponds to the creation of a new actor of class 'Actor$_i$' with a new address 'Adr', there is transition synchronised with this transition of the interface of 'Actor$_i$'.
- The transition 'replace(Act)': Is called by transitions associated to a 'become(Actor$_i$)' command. Then the actor 'Actor$_i$' is put in the place 'Actors'. The programmer must be aware of the problem of mutation in an other class of actors. Generally the become command is used in an actor of class 'Actor$_i$' to specify the replacement actor for the treatment of the next message in the buffer.
- The place 'Buffers' containing all the buffers of the instances of the class of actors 'Actor$_i$' (in the figure 18 'Buf$_{i,adr_k}$' and 'Buf$_{i,adr_l}$' are included in the place 'Buffers', they are respectively the buffer of address 'adr$_k$' and 'adr$_l$' of two different actors of same type 'Actor$_i$'.
- The place 'Actors' containing all the actor structures of the instances of the class of actors 'Actor$_i$' (figure 18 'Act$_{i,adr_k}$' and 'Act$_{i,adr_l}$' are included in the place 'Actors', they are respectively the structures of actor of address 'adr$_k$' and 'adr$_l$', they are two different actors of same type 'Actor$_i$'.

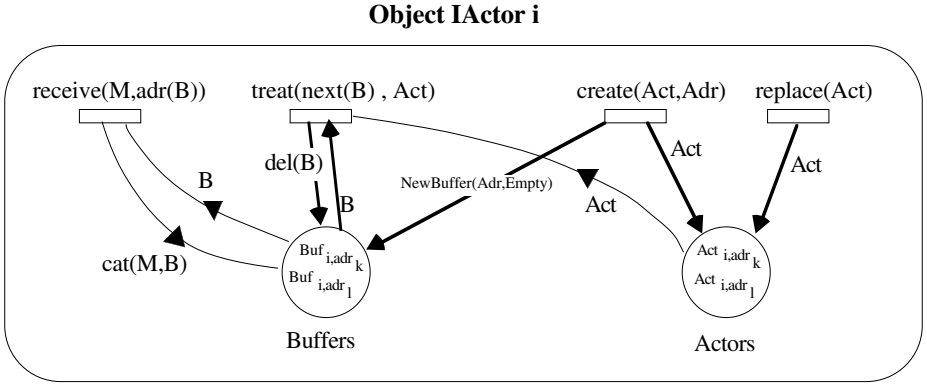

figure 18

We give below (figure 19) the generic specification of the object $IActor_i$ and its parameters using the CO-OPN formalism :

par : $BufferActor_i$;
use : Address, $EnrichMethodListActor_i$;
sorts : $bufferActor_i$;
generated by :
$bufferActor_i$: address * $methodListActor_i \rightarrow bufferActor_i$;
operations :
adr: $bufferActor_i \rightarrow$ address ;
axioms:...
end;

par : $MethodListActor_i$;
use : Address,$Actor_i$ Term ;
sorts : $methodListActor_i$;
generated by :
empty: $\rightarrow$ $methodListActor_i$;
cat : $Actor_i$Method * $methodListActor_i \rightarrow methodListActor_i$;
operations :
del: $methodListActor_i \rightarrow methodListActor_i$;
axioms:...
end;

par: $Actor_i$ Term;
sorts: $Actor_i$Method, $actActor_i$;
end;

gen object : $IActor_i$ ($BufferActor_i$, $Actor_i$Term);
use : Address ;
methods:
receive: $Actor_i$Method * address ;
treat : $Actor_i$Method * $actActor_i$;
create : $actActor_i$ * address ;
replace : $actActor_i$;
places : Buffers : $bufferActor_i$;
Actors : $actActor_i$;
var : M **:** $Actor_i$Method ; (sort declared in the specification $Actor_i$Term)
B : $bufferActor_i$; (sort declared in the specification $BufferActor_i$)
Act : $actActor_i$; (sort declared in the specification $Actor_i$Term)
Adr : address ;
axioms : receive (M,adr(B)) : Buffers([B]) $\rightarrow$ Buffers([cat(M,B)]) ;
treat(next(B)),Act) : not(is-empty(B))
$\Rightarrow$ Buffers([B]) , Actors([Act]) $\rightarrow$ Buffers([del(B)]) ;
create(Act,Adr) : $\rightarrow$ Buffers([$bufferActor_i$ (Adr,EmptyList)]) , Actors([Act]) ;
replace(Act) : $\rightarrow$ Actors(Act) ;
InitMarking : Buffers([]), Actors([]) ;
end;

Figure 19

3.4. Example illustrating the construction

In this section a simple actor program is presented which computes the function factorial. This example explains, step by step, the construction of the complete CO-OPN object system for the actor program given (figure 20).
Firstly, the algebraic specifications are constructed and secondly the objects for each actor are defined.

```
A1)  BegActor Factorial ()
A2)  [ do-it (m : address , n : nat , flag :bool)
A3)      become Factorial () ;
A4)      If (n = 1)
A5)      Then send reception (1) to m
A6)      Else   let x := new FactCust (m,n,flag)
A7)             send do-it (x , n-1 , false)
                to MyAdr
         fi ];
     EndActor;
```

```
B1) BegActor FactCust (  customer:address,
                         nb:nat, flag:bool)
B2) [ reception (value : nat)
B3)     If (flag)
B4)     Then        send Display(mult(nb,value))
                    to customer
B5)     Else        send reception ( mult(nb,value) )
                    to customer
        fi ];
    EndActor;
```

```
C1)  BegActor Global ()
C2)          [ Init ( )
C3)                  let p := new Factorial ()
C4)                  let q := new Factorial ()
C5)                  send do-it (MyAdr , 4 , true) to p
C6)                  send do-it (MyAdr , 6 , true) to q ];
C7)          [ Display (value : nat)
C8)                  Screen(value); ];
     EndActor;
```

Figure 20

This actor program is made of three actors which are 'Factorial','FactCust' and 'Global'. The computation of factorial is done with the following steps :

- Actors 'FactCust' with 'nb' to the value 'n' , 'n-1', ... '1' are created by the actor 'Factorial'.
- After the creation of the last actor 'FactCust' with 'nb'=1, this one start the process where the i^{th} actor sends back the product of i by the product coming from the previous actor which is equal to (i-1)!
- The first actor sends the result to the actor 'Global'.

A short description of the computation of 4! is presented in figure 21 :

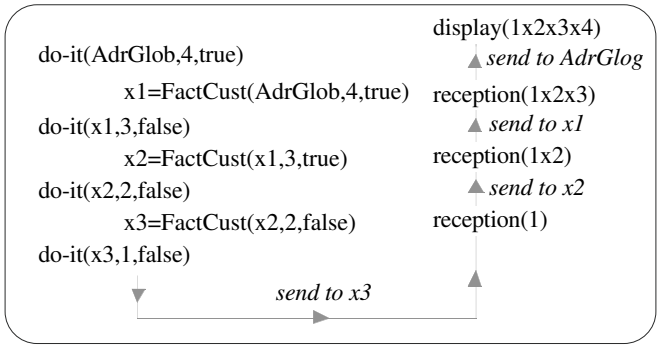

figure 21

One can note that this computation is quite sequential. But, in the actor 'Global' two computation can be started concurrently (for instance 4! and 6!) . The detail of the algebraic specification and the set of CO-OPN objects built for the actor 'Factorial' are given below.

Algebraic specifications associated to actor 'Factorial'
Each actor has a buffer which is a Fifo structure. The first part of this paper has introduced a generic algebraic specification for Fifo. Each actor having its buffer, the specification module for the buffer of the actor 'Factorial' is :

```
spec MethodListFactorial
as Fifo (Elem ⇒ Factorial_Term by morph)
where morph : elem ⇒ FactorialMethod
renaming fifo into methodListFactorial
end;
```

Figure 22

```
spec BufferFactorial
as CP (E1 ⇒ Address by morph1,
       E2 ⇒ MethodListFactorial by morph2)
where morph1 : e1 ⇒ address;
where morph2 : e2 ⇒ methodListFactorial;
renaming cp into bufferFactorial; p1 into adr;
end;
```

Figure 23

The specification of the cartesian product of two sorts 'CP' is the classical one with two operations P1,P2 corresponding to the projections.

The examination of the actor programs permits to enumerate all the sorts needed. For each mentionned sort there must be an associated algebraic specification; for example, the types address, nat and bool are used in the actor 'Factorial' with the corresponding instruction '**use** : Address,Nat,Bool'.
The new sorts which are declared in the algebraic specification module of the actor 'Factorial' are 'actFactorial' representing the data structure associated to the actors 'Factorial' and 'factorialMethod' representing the possible messages for the actor 'Factorial'.

The generators of the sort 'actFactorial' is '<_,_,_,_,_>: address * address * nat * bool * address → actFactorial', representing the data structure associated to this actor. Then an actor 'Factorial' is represented as a 5-tuple of values. The first value is of the sort address and represents the address of the actor. All the others are the values of variables encountered while parsing the code of the actor 'Factorial'. The second is for 'm' (sort address), the third for 'n', the fourth for 'flag' (sort bool) and the fifth for 'x' (sort address).

For example, if an actor has the address 5 and if the value of (m,n,flag,x) are (6,2,true,7) respectively, then the actor is <5,6,2,true,7> and is a value of the corresponding algebra of the sort 'actFactorial'.

The generators of the sort 'factorialMethod' are the name of the methods declared in the actor 'Factorial'. The operations associates to this sort are projections (P1_,P2_,..) giving the parameters of the n-tuple of values.

Only one method 'do-it' is declared in this actor. The generator of the sort 'factorialMethod' is 'do-it_ _ _ : address * nat * bool → factorialMethod' this allow to build messages as algebraic terms.

```
spec :  Factorial_Term;
use :   Bool, Nat, Address;
sort :  actFactorial,factorialMethod;
generated by :   <_,_,_,_,_> : address address * nat  * bool  * address → actFactorial;
                 do-it _ _ : address *  nat → factorialMethod ;
operations :     P1_: actFactorial → address;     (*gives the first value *)
                 P2_ : actFactorial → address ;   (*gives the second, etc…*)
                 P3_ : actFactorial → nat ;
                 P4_ : actFactorial → bool ;
                 P5_ : actFactorial → address ;

axioms :    Proj1 :     P1(<adr1,adr2,n1,b1,adr3>) = adr1 ;
            Proj2 :     P2(<adr1,adr2,n1,b1,adr3>) = adr2 ;
            ……
where :     adr1,adr2,adr3: address;
            n1 : nat;
            b1 : bool;
end;
```

Figure 24

Object specification of the actor 'Factorial'

Even if the type address is not used in an actor code it is added at the '**use**' declaration because of the use of the variable 'MyAdr'. This variable is interpreted as the actor address and can be utilised everywhere in an actor code. The CO-OPN specification of the actor 'Factorial' is made of the specification of the method 'do-it' and of the interface 'IFactorial' obtained by instanciating 'IActor$_i$' (see the figure in the annex showing the CO-OPN specification, the bold arcs linking transitions are the CO-OPN synchronization, only the synchronization of the object 'do-it' are shown). The other objects are done in a similar way and represented on the same design)

```
object :    IFactorial as IActor_i (BufferActor_i => BufferFactorial
            by morph1,  Actor_iTerm => Factorial_Term by morph2);
where :     morph1: bufferActor_i => bufferFactorial;
            morph2:     Actor_iMethod => factorialMethod;
                        actActor_i => actFactorial;
end;
```

```
1)  object :    do-it ;
2)  use :       IFactorial ,  IGlobal, IFactCust,
                Factorial_Term, Global_Term, Fact_CustTerm, Address,Nat,Bool ;
3)  methods :   do_it : address * nat * boolean;
4)  internal :  t2,t3,t4,t5,t6,t7,end;
5)  places :    pl1,pl2,pl3,pl4,pl5,pl6: actFactorial;
6)  var :       x,cpt : address; p:nat;fl : bool;
    axioms :
7)      do_it with treat(do_it(x,p,fl),Act)              : → pl1([<P1(Act),x,p,fl,P4(Act)>])
8)      t2     with replace(Act)                         : pl1([Act]) → pl2([Act])
9)      t3                                               : P3(Act) ≠ zero ⇒ pl2([Act]) → pl4([Act])
10)     t4                                               : P3(Act) = zero ⇒ pl2([Act]) → pl3([Act])
11)     t5     with receive(reception(1),P2(Act))        : pl3([Act]) → pl6([Act])
12)     t6     with sim(create(<cpt,P2(Act),P3(Act),false,zero>,get_new_address(cpt)),P2(Act))
                                                        :pl4([Act]) → pl5([Act])
13)     t7     with receive(do-it(x,n-1,false),P1(Act)) pl5([Act]) → pl6([Act])
14)     end                                              : pl6([Act]) →
15) init_marking:
    end;
```

Figure 25

We explain some CO-OPN declarations of the object 'do-it' (figure 25):

2) The actor 'Factorial' must declare the algebraic specification used by its related actors ('Factorial_Term' , 'FactCust_Term', 'Global_Term'). Furthermore, the interfaces corresponding to the actor used by 'Factorial' are declared ('IFactorial', 'IFactCust', 'IGlobal').
3) The methods are shown in the transitions system built for the semantics of the total CO-OPN system.
4) The internal transitions are not shown in the transition system.
5) All the places are declared of sort 'ActFactorial' to contain the actors of class 'Factorial'.
7) The method 'do-it', is synchronised '**with**' the method 'treat' of the object 'IFactorial'. The parameters of the method 'treat' indicate that the place 'Buffers' must have a message 'do-it(m,n,flag)' and the place 'Actors' must contain the associated actor (i.e the actor to which the buffer belongs). If such synchronization is possible then the transition is fired, corresponding to the beginning of the method 'do-it' and the actor variables are updated .
12) The new address needed by the creation of a new actor is given with the method 'get_new_address'.

4. DISCUSSION ABOUT THE MODELLING OF ACTORS USING CO-OPN

In this section the concurrency aspects of our modelling of actor languages are discussed and some proposal to improve it are suggested.

Concurrency between actors

Of course, all instances of actors allow concurrent behaviour. This is modelled by the possible concurrent firing of the interface methods expressed by the CO-OPN semantics. But these firings can be sequentialized by the message treatment. The concurrency between actor instances is modelled by the concurrency between CO-OPN objects and by the concurrency inside CO-OPN objects.
Note that the dynamicsof the actors creation is modelled by the creation of tokens in the interface objects.

Concurrency on address creation

A sequentialization appears in the creation of actor address, because the object 'NewAdress' cannot be accessed concurrently. Two solutions appears : The first one consists of keeping a centralized address manager which can be concurrently accessed by several actors (see the object 'NewAddress' with a prime number property for the generation of unique address). The second one consists of distributing the address manager, using a separated one for each actor class. These improvements are related to the communication costs and parallelism degree. With this new definition of 'NewAddress' three unique addresses can be distributed concurrently (even more with the same principle).

object : NewAddress ;
use : Address,CPAddress;
methods : get_new_address:address ;
places : next_address: address;
var : m:address;
axioms : get_new_address(m):
next_address([cp(n , m)]) → next_address([cp(n , mul(m,n))])
init_marking : next_address([cp(two,two),cp(three,three),cp(five,five)]);
end;

Concurrency inside a method
The execution of the instructions of one method are sequentialized. This is due to our kind of actor language. The data structures dependencies allow to parallelize the method code. This can be modelled by CO-OPN using the appropriate Petri net design.

Serial actor and unserial actor
Our actor model is a serial actor model. If unserial actors are considered, our semantics should allow concurrent access to buffers. This is done by the concurrent firing of the transitions 'receive' or 'treat'. Furthermore, the messages have to be in the multiset of a place instead of in the Fifo structure.

Fairness
A condition for the right execution of actor program is that the execution of transitions should be fair. This condition is not an usual property of Petri nets. Thus fairness has to be respected in the execution of the CO-OPN system.

Actor replacements, actor equivalence
Within the framework of CO-OPN formalism it is possible to describe the actor method replacement with CO-OPN object replacement. To insure an actor replacement, the old actor method and the related interface object have to be equivalent (w.r.t the definition of bisimulation **[B&G 90]**) to the newest one with the same interface object. This replacement can be used to implement such actor programs using CO-OPN refinements.

5. CONCLUSION

Our modelling of actors using CO-OPN takes into account both data structures and modularity. The data structures are modelled using algebraic specifications, the modularity using CO-OPN objects and communication between actors through CO-OPN synchronization expressions and actors ' interfaces.
The contribution of our formalism to the modelling of actors is new, important and twofold:
- The ability to obtain a semantics of actors including data structures, modular aspects, and true concurrency.
- The possibility to model concurrent object oriented languages.

Further work can then be divided in two parts :

- Using the modular properties of CO-OPN formalism to analyze **[R&B 92]** and to refine such specification (the tools actually under development or developed are used to analyze and to prototype **[B&F&R 92]** actor specification).
- Improving CO-OPN with inheritance mechanism and object class structuring.

ACKNOWLEDGMENTS

We are especially grateful to Elisabeth Pelz, Stephane Boucheron and Nicole Bidoit for their helpful suggestions on this paper.

BIBLIOGRAPHY

[AGHA 86] **G. Agha:** "Actors: a model of concurrent computation in distributed systems", The MIT Press (1986)

[B&F&R 92] **D. Buchs, J. Flumet, P. Racloz** "Producing prototypes from CO-OPN specifications", 3rd IEEE International workshop on Rapid system Prototyping, Research Triangle Park, USA, June 1992.

[B&G 90] **D. Buchs, N. Guelfi** "CO-OPN : A Concurrent Object Oriented Petri Nets Approach for System Specification", internal report LRI n°616, UPS Orsay, 1990

[B&G 91A] **D. Buchs, N. Guelfi** "CO-OPN : A Concurrent Object Oriented Petri Net Approach for System Specification", 12th International Conference on Application And Theory of Petri Nets, Aarhus 1991.

[B&G 91B] **D. Buchs, N. Guelfi** "Distributed System Specification using CO-OPN", 3rd IEEE International workshop on Futur trends of Distributed Computing Systems in the 90 's, Taipei 1992.

[B&G 91C] **D. Buchs, N. Guelfi** "Open distributed programming using the object oriented specification formalism CO-OPN", 5th International conference on Putting into Practice Methods and Tools for information system Design, Nantes 1992.

[B&G 91D] **D. Buchs, N. Guelfi** "A Semantic Description of Actor Languages using CO-OPN", 1 Page abstract in the 14th International Conference On Software Engeeniring, Melbourne, 1992

[B&G&M 91] **D. Buchs, N. Guelfi, F. Mourlin** "Semantic description of OCCAM-2 language using CO-OPN", DEMON Worshop, Aarhus 1991.

[BATT 89] **E.Battiston, F.De Cindio, G.Mauri** "OBJSA a class of High level nets having objects as domains", LNCS n°340, pp 20-43

[BIDO 87] **M. Bidoit** "The stratified loose approach : a generalization of initial and loose semantics", Recent Trends in Data Type Specification, selected papers of the 5th work-shop on specification of abstract data types, LNCS 322 pp 1-22, 1987.

[BIDO 89] **M. Bidoit** "PLUSS, un language pour le developpement de spécification algébrique modulaires" Thèse de doctorat d'Etat, LRI UPS Orsay

[BUCH 88] **D. Buchs,** "EM2/specif : a method for specifying interactive application", Workshop on software engineering and its application, pp 540-556, Toulouse dec. 1988.

[CLIN 81] **W.D. Clinger** "Foundations of actors semantics", AI-TR-633, MIT artificial intelligence laboratory, (1981).

[D&J 90] **N. Dershowitz and J.P. Jouannaud** "RewriteSystems" In Handbook of Theoretical Computer Science, Vol. B, North-Holland, 1990.

[E&M 85] **H.Ehrig, B.Mahr** "fundamentals of algebraic specification 1: equations and initial semantics,Springer Verlag 1985."

[ENGE 90] **J. Engelfriet, G. Leih, G. Rozenberg** " Parallel object-based systems and Petri nets", Technical Reports 90-04 and 90-05, Leiden University, 1990.

[FIAD 90] **J.Fiadero, T.Maibaum** "Describing, structuring and implementing objects", in J.de Bakker, W de Roever and G.Rozenberg (Eds) Foundations of objects oriented languages, Springer Verlag.

[GOGU 78] **J.A. Goguen, J.W. Thatcher, E.G Wagner** "An initial algebra approach to the specification, correctness and implementation of abstract data types", in current trends in programming methodology IV:Data structuring,Prentice Hall,1978,80-144

[GUEL 90] **N.Guelfi** "Classes de réseaux structurés : étude, utilisation et modification." Rapport de stage de DEA 90, LRI UPS Orsay (1990)

[HEWI 73] **C.E. Hewit, P. Bishop, and R. Steiger** A Universal Modular ACTOR Formalism for Artificial Intelligence. In proceedings of the 3rd IJCAI, pages 235-245, Stanford, California, 1973.

[HOAR 85] **C.A.R. Hoare** "Concurrent Sequential Processes", Prentice Hall, 1985.

[HOPK 89] **R.P. Hopkins, M. Koutny, B.Randell** "Some results on dynamically structures communicating systems", research memorandum, University of Newcastle upon Tyne (1989)

[JANS 89] **D.Janssens & G.Rozenberg** "Actors grammars", Math. syst. Theory 22, 75-107, (1989).

[JENS 81] **K. Jensen** "Coloured Petri Nets and the invariant method", TCS 14 , pp 317-336, North Holland Pub.Co (1981)

[KRAM 87] **B.Krämer** "Segras : a formal language combining Petri nets and abstract data types for specifying distributed systems" in proceedings of the 9th annual international conference on software engineering, pp 116-125, Monterey, California March 1987.

[LOGR 88] **L.Logrippo, and Al :** "An interpreter for Lotos : A specification language for distributed systems", in software practice and experience, vol. 18 n°4, 1988.

[MILN 89] **R. Milner** "Communication and Concurrency". Prentice Hall, 1989.

[NIER 90] **O. Nierstrasz** "A guide to Specifying Concurrent Behaviour with Abacus", in Object Management (ed) D. Tsichritzis, University of Geneva, 1990.

[POME 86] **L. Pomello** "Some equivalence Notions for Concurrent Systems: An Overview". In: Advances in Petri nets 1985 Springer Verlag, LNCS 222 pp. 381-400.

[R&B 92] **P. Racloz, D. Buchs** "Symbolic Proof of CTL formulae over Algebraic Nets", to appear as CUI internal report, CUI Geneva University, 1992

[REIS 85] **W. Reisig** "Petri Nets" Springer Verlag (1985)

[SANN 85] **D.Sannella, A. Tarlecki** "On observational equivalence and algebraic specification", TAPSOFT 85, Springer verlag, LNCS 185 pp308-322.

[V&S 90] **G. Vidal-Naquet, Y. Sami** "Formalisation of the behavior of actors by colored Petri nets and some applications", PARLE 91, LNCS N°506.

[VAUT 87] **J.Vautherin** "Parallel systems specification with coloured Petri nets and algebraic specification" LNCS ICPN 87

[YOKO 86] **Y.Yokote, M.Tokoro** The Design and Implantation of Concurrent Smalltalk, SIGPLAN Notices, Vol 21, No 11, 1986.