CO-OPN for specifying a distributed termination detection algorithm

D. Buchs, F. Mourlin, L. Safavi

CUI, University of Geneva
24 rue du General Dufour
1211 Geneva 4, Switzerland

buchs@cui.unige.ch
mourlin@cui.unige.ch
safavi@cui.unige.ch

Abstract

The SANDS environment along with the CO-OPN specification formalisms are presented. The latter permitting specification of data types and concurrency through the use of modular algebraic specifications and structured algebraic nets based upon an object oriented approach. System specification will be demonstrated through an ascending distributed termination detection algorithm. Major aspects of the environment and the CO-OPN formalism will be discussed.
Key words: Concurrent specification, structured algebraic nets, distributed termination detection.

I Introduction

CO-OPN (Concurrent Object-Oriented Petri Net) has been designed in order to obtain a complete specification formalism for complex concurrent systems and languages. This aim has been reached by using a modular algebraic specification language connected with a concurrency part in terms of nets. The algebraic specification language is used in order to define data structures needed in the specification of the system. In other words, we use algebraic nets as modules of the specification to describe the system in a structured way. CO-OPN can be seen as an extension algebraic nets including modularity by means of a simple notion of objects. We adopt the object oriented approach by defining methods for encapsulation and synchronization expressions for abstraction, within an object. An object has an internal and an external behavior. Internally, the algebraic net provides system evolution, whereas the external behavior, relying on methods, provides an interface with synchronization expressions to describe the call of other methods. Method calls can be viewed as procedure calls. Symbolic links, which are oriented, represent method calls.

The semantics of CO-OPN is defined by introducing the notion of distributed transition systems (named DTS hereafter) [7]. For a given CO-OPN specification, its DTS is built using inference rules over object markings. DTS transitions correspond to a vector of multi-sets of events, one component for each object of the specification. It is to be noted that the stepwise construction of the DTS by inference rules follows the object hierarchy induced by synchronization expressions.

A formal framework of specification refinement has been defined in order to derive an implementation from an abstract CO-OPN specification. CO-OPN specification equivalence exists if a bisimulation relation relates two different specifications. In a given system, an object can be replaced by another if they are bisimilar. The bisimulation between two different DTSs is defined for the same algebra on both systems. A second refinement notion is based on observationally equivalent algebras allowing change to data implementation without changing the behavior of the system.

In this paper a specification, analysis and implementation of a distributed termination detection (DTD) algorithm will be given. The latter will be exposed algorithmically and specified using CO-OPN specification formalisms. The SANDS (Structured Algebraic Net Development System) environment allows graphical representations, simulations and system proving and proof through temporal logics.

The analysis will show that CO-OPN along with SANDS are well suited for distributed system specification. The adequacy of the latter formalism has also been shown in other papers concerning languages such as OCCAM2 and Open distributed Systems [6].

II The DTD Algorithm

In distributed systems, where processes communicate solely by messages, the detection of termination of a distributed computation is non-trivial since no process has complete knowledge of the global state. The distributed termination was founded by N. Francez [9]. A distributed computation is considered to be globally terminated if every process is locally terminated and no messages are in transit. Local termination can be understood to be a state in which a process has finished computation and will not restart any action unless it receives a message. The aim of DTD algorithms is to detect global termination by means of additional control messages.

In the analyzed algorithm within this paper an ascending detection scheme has been adopted by its authors. It does not rest on the observation waves principle. It is to be noted that the control and base algorithms are distinct entities running at the same time for each node of the system. The control is achieved with a rooted tree pattern associated to an ascending routing of colored tokens. The leaves start the detection operation, which is completed by the root. The algorithm is presented without the Fifo requirement within the context of an asynchronous message passing scheme. It can be easily adapted to a synchronous communication scheme with or without a fifo requirement for message reception and transmission.

The algorithm is based on a rooted spanning tree where each node represents a process. Each of the latter processes knows the identity of its father node and the list of all its direct sons in the tree with their color. Three colors, red, white and blue designate the state of each process during the termination detection phase. Messages transit through fault free channels and no message duplication is assumed.

Initially at start up, an initial process, known as the root, initiates the control network, creating the spanning tree of control, which can be different than the base topology of the network. All processes are initiated active and blue. The leaves start termination detection. As a node becomes white and idle, it announces its state to its father. This process continues until the root receives knowledge that all its sons are white and idle. Termination is then assumed. The algorithm is presented below.

II.1. Algorithm description

The types of messages and data structures used are as follows.
Color is in the set [red, white, blue], control and basic messages transit through the control topology and the base topology respectively. *Basic messages are* acknowledged and delivered through the base topology. Each process is aware of its direct father, *Father*, in the control topology. The identities of the sons of a node are known, denoted *Son*, **i**n the control rooted tree. If the node is a leaf, then *Son* = ∅. *My_State* represents the state of node i and belongs to the set (active, idle). Initially *My_State* = active. *Nack*, an integer value, represents the number of acknowledgments, named *Ack messages*, i is waiting for. Initially *Nack* = 0. *C[i]*, denoted *NodeColor* in the specifications, is the color of node i and belongs to the set [red, white, blue]; initially *C[i]* = blue. For each k belonging to the set *Son*, *C[k]* which is the color of k at i

belongs to (red, white, blue); initially *C[k]* = blue. *Dest* is the neighbor if i which destabilized it by sending a basic message; initially *Dest* = nil. *Dest_Son* is the list of the sons of i, in the base topology, which destabilized it by sending a red token; initially *Dest_Son* = ∅.
The list of rules adopted by the algorithm, in the case of asynchronous exchange of messages without a fifo requirement about the transmissions channels, is as follows.
Rule 0: Initially every process is active and blue; each channel is empty.
Rule 1: when a process receives a basic message
If it is white then it memorizes the sender as a destabilizer
it becomes red
it sends a red token to its father
else it acknowledges the message
Rule 2: a blue process may become white if and only if
it is idle
all its outgoing channels are empty
all its sons are white
Rule 3: when a process becomes white, it sends a white token to its father.
Rule 4: when a process receives a red token from a son k
if it is white then it memorizes k as a destabilizer
it becomes red
it sends a red token to its father
if it is blue it sends to k a blue token
if it is red it memorizes k as a destabilizer
Rule 5: when a red process receives a blue token
it becomes blue
it sends a blue token to the sons which destabilized it
it acknowledges the other destabilizer, if it exists

II.2. Topologies of the algorithm
Figure 1 depicts a base topology over which a control topology has been constructed. The base topology is a bidirectional ring structure whereas the control topology represents a rooted tree with N0 as its root.

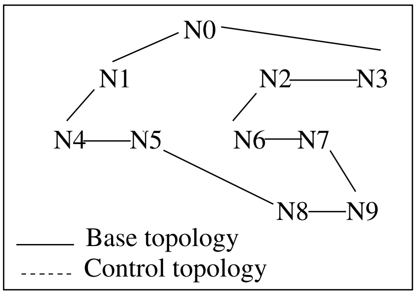


figure 1 Example of base and control topologies

The basic messages, and acknowledgments, transit through the base topology. Color tokens and control messages transit through the tree structure. We can note that basic messages can be acknowledged by a node which is not a neighbour node in the control topology.

III The Algebraic specification
A CO-OPN specification is composed of two parts. An algebraic specification describing the data structures used in the system and a set of objects (algebraic nets) using the algebraic spec-

ification and specifying the overall system. The complex criteria is to quantify how much of the overall system should be represented by algebraic nets and how much of it should be dealt with through algebraic specifications. This paper does not attempt to deal with the latter consideration but to give a global view of specification construction. We will give a general view of algebraic specification modules and their underlying sorts.

The notion of hierarchy is adopted for specifications as well as for CO-OPN objects.

III Algebraic Specification

A CO-OPN specification is composed of two parts. An algebraic specification describing data structures used in the system and a set of objects, algebraic nets, using the algebraic specification and specifying the overall system. The algebraic specification part of system specification should be undertaken in consideration of the different data structures needed. In our example of DTD the need for specifying an abstract data type representing node states, named ‘NodeState’, is the most obvious. Different abstract data types such as ‘Nat’, denoting natural numbers, ‘Bool’, denoting the booleans, are also immediate types needed in order to be able to expand description of the algebraic specification of the overall system. Upon these structures more concise and more specific structures are then built, namely ‘List’, which takes use of the predefined data types in order to derive a new type definition, see figure 2.

SPEC Bool;
SORTS bool;
GENERATED BY
true : -> bool;
false : -> bool;
OPERATIONS
not : bool -> bool;
and : bool -> bool;
or : bool -> bool;
xor : bool -> bool;
equal : bool -> bool;
AXIOMS
....
END

SPEC Nat;
USE Bool;
SORTS nat;
GENERATED BY
zero : -> nat;
succ : nat -> nat;
OPERATIONS
add : nat, nat -> nat;
sub : nat, nat -> nat;
mul : nat, nat -> nat;
div : nat, nat -> nat;
mod : nat, nat -> nat;
decr : nat -> nat;
equal : nat, nat -> nat;
AXIOMS
....
END

PAR SPEC Item;
USE Bool;
SORTS item;
OPERATIONS
equal : item , item -> bool;
END

GENERIC SPEC List(Item);
USE Bool, Nat, Item;
SORTS list;
GENERATED BY
emptylist : -> list;
cons : item , list -> list;
OPERATIONS
cat : list, list -> list;
length : list -> nat;
take : nat, list -> list;
drop : nat, list -> list;
head : list -> item;
tail : list -> list;
is_empty : list -> bool;
reverse : list -> list;
append : list, item -> list;
AXIOMS
....
END

figure 2 Algebraic specification of booleans, natural numbers, generic list specification and parameter specification

It remains, however, to distinguish which part of the system is to be specified using algebraic specifications and which part with algebraic nets. There is no answer to this criteria. We believe that the net is well suited for the main actions with their execution order relations, while the algebraic specification part may be used to specify basic data structures with their associated operations. A CO-OPN specification could be made of one object with one place and a transition computing an algebraic term representing the system behavior. In this case all the specification is made with the algebraic part only. On the opposite, we could imagine a CO-OPN specification with an algebraic part reduced to the minimum and substituted by net constructions. These two extremes are not realistic. They only give an idea of the decomposition problem of a specification. In this paper we will not deal with the latter considerations but only deal with the general principles of specification construction. We will discuss algebraic specification modules and their underlying sorts. We will also give a partial description of the specification of sorts used. The axioms[1], not given, describe the properties of the operations associated to the specification. They are useful for testing, refin-

1. Reserved words in the CO-OPN language are in bold characters.

ing and proving specifications.

III.1. Basic sorts

In our distributed termination detection algorithm another obvious structure needed to be represented are the messages, 'Message', in transit within the communication links between processes. Node states are crucial aspects of the process states during termination detection; a process being in the set [active, idle]. In the control structure another important aspect is to be able to identify each process, or node, individually. As a result type 'NodeId', which represents identifications in natural numbers is introduced. 'NodeId' is said to refine sort 'Nat', see figure 3. The list of sorts presented here is not complete and is alleged in order to put through some general points over specification description.

SPEC Message;
USE Bool;
SORTS message;
GENERATED BY
ackM : -> message;
cstM : -> message;
emptyM : -> message;
OPERATIONS
eqM : message, message -> bool;
AXIOMS
......
END

SPEC NodeColour;
USE Bool;
SORTS nodecolour;
GENERATED BY
red : -> nodecolour;
white : -> nodecolour;
blue : -> nodecolour;
OPERATIONS
eqNC : nodeColour, nodeColour -> bool;
AXIOMS
....
END

SPEC NodeId **AS** Nat;
RENAMING
nat **INTO** nodeId;
equal **INTO** eqN;
END

SPEC NodeState**;**
USE Bool;
SORTS nodeState;
GENERATED BY
idle : -> nodeState;
active : -> nodeState;
OPERATIONS
eqNS : nodeState, nodeState -> bool;
AXIOMS
...
END

figure 3 Basic sorts

III.2. Generic specifications

The notion of hierarchy is adopted for specifications as well as for CO-OPN objects. Initially the so called fundamental sorts, such as 'Nat', 'Bool' (see figure 2) are constructed. Then generic sorts, such as 'List', with their parameter specifications, 'Item,' are introduced. A generic specification is an algebraic specification associated to a parameter list. Each parameter is the name of a parameter specification. A parameter specification, 'Item', defines the sorts that can be used in the related generic specification 'List'. The generic specification of 'List' describes the set generated by the operators 'emptylist' and 'cons'. The latter uses the basic specifications 'Bool, 'Nat' and the parameter specification 'Item'. Parameter sorts for the generic specification are defined to give a model to the parameters.

'Pc3' , see figure 4, represents the cartesian product of 3 sorts. The projections on a

GENERIC SPEC Pc3 (Item1, Item2, Item3);
USE Bool, Item1, Item2, Item3;
SORTS pc3;
GENERATED BY
triplet : item1, item2, item3 -> pc3;
OPERATIONS
first : pc3 -> item1;
second : pc3 -> item2;
third : pc3 -> item3;
equal : pc3, pc3 -> bool;
AXIOMS
....
END

GENERIC SPEC Fifo (Item);
USE Bool, Nat, Item;
SORTS fifo;
GENRATED BY
emptyfifo : -> fifo;
push : item, fifo -> fifo;
OPERATIONS
pull : fifo -> fifo;
length : fifo -> nat;
top : fifo -> item;
is_emptyF : fifo -> bool;
AXIOMS
....
END

Figure 4 Algebraic specification of generic sorts

chosen component are given with the functions 'first', 'second', 'third'. The 'Pc3' sort is useful to define the basic structure of the messages which are exchanged between nodes. 'Fifo' represents a first in first out buffer structure.

III.3. Instantiated sorts

Specification 'PcSon' is an instantiated sort composed of an eight cartesian product, 'Pc8', over different basic and generic specifications, see figure 5. These sorts are constructed over the said primitive or basic sorts. Examples of such sorts are 'BasicMessage (Message,NodeId,NodeId)', a 'Pc3' of three basic sorts, and 'FifoMessage (Fifo, Message)', a 'Pc2' of 'Fifo' and 'Message'.

```
SPEC PcSon AS Pc8 (NodeId, Father, ListSon, State, Nack, NodeColour, Dest, DestSon);
RENAMING
pc8 INTO pcSon;
first INTO prjNI; second INTO prjF; thrid INTO prjL; fourth INTO prjST; fifth INTO prjN; sixth INTO
prjNC;
seventh INTO prjD; eighth INTO prjDS;
USE NodeId, Father, ListSon, State, Nack, NodeColour, Dest, DestSon;
SORTS pcSon;
OPERATIONS
destab:          pcSon, nodeId -> pcSon;
acknow :         pcSon, nodeId -> pcSon;
blueson :        pcSon, nodeId -> pcSon;
decr_nack :      pcSon -> pcSon;
is_white :       pcSon -> pcSon;
is_red :         pcSon -> pcSon;
is_blue :        pcSon -> pcSon;
sendAck :        pcSon -> pcSon;
go_idle :        pcSon -> pcSon;
sendFather :     pcSon -> pcSon;
whiteToken :     pcSon, nodeId -> pcSon;
redToken :       pcSon, nodeId -> pcSon;
blueToken :      pcSon, nodeId -> pcSon;
blueDest :       listSon, destSon -> destSon;
allWhiteSons :   pcSon -> bool;
nonack :         pcSon -> bool;
is_idle :        pcSon -> bool;
terminate :      pcSon -> pcSon;
AXIOMS
 .......
END
```

Figure 5 Instantiated sort PcSon

Once the whole system has been specified algebraically, a hierarchical dependency relationship is established between the abstract data types. Figure 6 shows a series of specifications, instantiated, which will be of need in the following of the paper. They will be used in the correctness proving within section V.

```
SPEC ListSon AS List (SonFeatures);
RENAMING list INTO listSon; emptylist INTO noSon; cons INTO addSon; head INTO firstSon; tail INTO otherSon;
USE Bool, DestSon, NodeColour, NodeId, SonFeatures;
OPERATIONS
become_blue : listSon, nodeId -> listSon;
become_white :listSon, nodeId -> listSon;
become_red : listSon, nodeId -> listSon;
blueDest : listSon, destson -> listSon;
belongs_to : listSon, nodeId -> bool;
AXIOMS
.....
END

SPEC FifoCtrlMsg AS Fifo(CtrlMessage);
RENAMING fifo INTO fifoCtrlMsg; emptyfifo INTO emptyCtrlMsg; push INTO pushCtrlMsg;top INTO firstCtrlMsg;
is_emptyF INTO is_CtrlMsg;
USE CtrlMessage;
END


SPEC OwnCtrlMsg AS Pc2(NodeId, FifoCtrlMsg);
RENAMING pc2 INTO ownCtrlMsg; fst INTO ctrlOwner; snd INTO ctrlFifoBox;
USE NodeId, FifoCtrlMsg;
END


SPEC CtrlMessage AS Pc3 (NodeColour, NodeId, NodeId);
RENAMING pc3 INTO ctrlMessage; fst INTO ctrlColour; snd INTO ctrlTransmit; thrd INTO ctrlReceive;
USE Bool, NodeColour, NodeId, Nat;
END

SPEC FifoMessage AS Fifo(BasicMessage)
RENAMING fifo INTO fifoMessage; emptyfifo INTO emptyBox; push INTO post;top INTO firstMessage, pull INTO read;
is_emptyF INTO is_noMessage;
USE BasicMessage;
END

SPEC OwnMessage AS Pc2(NodeId, FifoMessage);
RENAMING pc2 INTO ownMessage; fst INTO owner; snd INTO fifoBox;
USE NodeId, FifoMessage;
END
```

Figure 6 Instantiated sorts

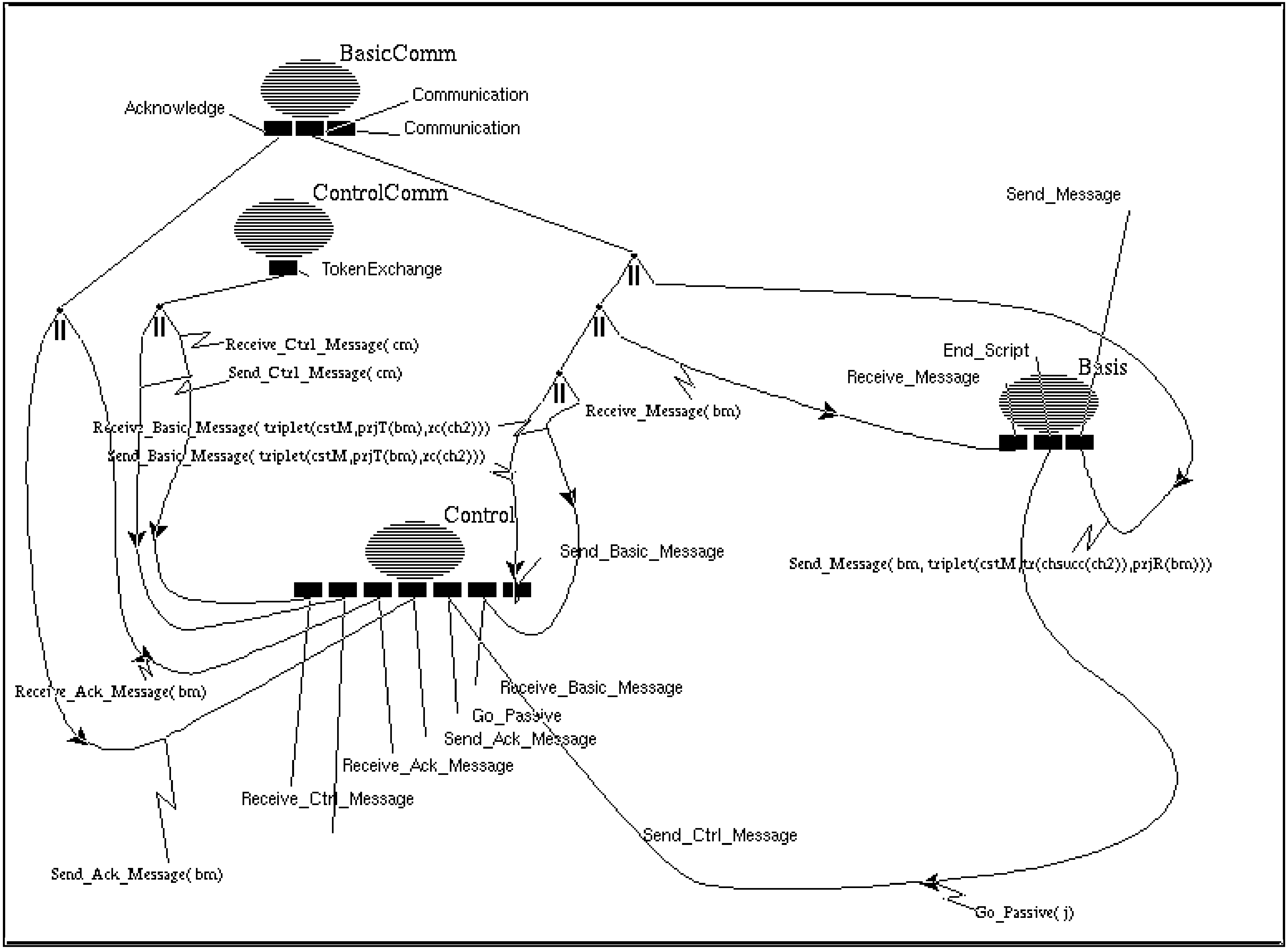
IV The CO-OPN specification

The first step in defining the CO-OPN system is to determine the set of concurrent objects which constitute the system. The system is then structure in different and independent segments, named objects. Each object will then have externally visible interfaces, named methods, which will permit synchronization with the other objects of the overall system. Objects evolve internally through places and internal transitions. It is to be noticed that the concurrency of the overall system exists at the method level as well as at the internal behavior of each individual object. The CO-OPN specification is then completed with synchronization links between objects. The latter can be of three types: 'sequential', 'parallel' or 'alternative'. These synchronization link construct constraints upon the overall system objects.
The overall algorithm has been modelled into four distinct objects shown in figure 3. Ellipses represent objects composed of algebraic nets, arcs are synchronization links, grey rectangles represent methods and internal transitions within objects are represented by white rectangles. Object places are symbolized through circles.
The objects Basis, Basic_Comm, Control, Control_Comm and their respective methods will be detailed individually.
It is to be noted that the base algorithm and control algorithm are two different and distinct entities. The base algorithm is dedicated to perform a certain amount of computation and communication, the control algorithm has as sole activity to detect termination based upon the state of each node. As a result each node possesses a base and a control part running. The distinction between the two entities is presented within the object distribution as two different objects.

The different objects involved in the overall specification are diagrammatically shown with their respective hierarchical positions, see figure 2. The two communication objects, namely Basic_Comm and Control_Comm are destined to represent the base and the control topologies. Object Basic_Comm represents an oriented graph containing the configuration of the base topology while object Control_Comm has knowledge of the rooted spanning tree. The two objects deal with delivery and reception of messages sent from one node to another.
Object Basis deals with the state and messages of a node. Object control provides the mechanism of controlling
the state modifications of a node and assures termination detection at local and global levels.

IV.1. Object Basis
The system modelled represents a collection of n concurrent processes, with n being a finite integer. Each process is in either of four states: computing, sending or receiving messages from other processes or inactive. In the latter case it will become white and announce its idle state to its father node. Object Basis has two places, namely Activity and Buffer. Activity holds n tokens of type `Script`, one for each node of the system. Each token is a script defining each node's behavior, see figure 7.

Place Buffer holds n tokens each of type `OwnMessage` for each node of the system where incoming messages are stored. `OwnMessage` is an algebraic specification composed of a cartesian product of two sorts: the node identity (a natural number) and a fifo buffer for message reception in an asynchronous transmission mode. Methods Send_Message and Receive_Message deal with incoming and outgoing messages and possess parameters of type BasicMessage, namely a triplet composed of a message, a transmitter node identity and a receiver node identity. At local termination detection, method End_Script synchronizes with method Go_Passive of the control object in order to update the state of the idle node in the control data structure.
Two internal transitions Computation and UsedMessages evolve the object internally.



SANDS generated Co-opn objects

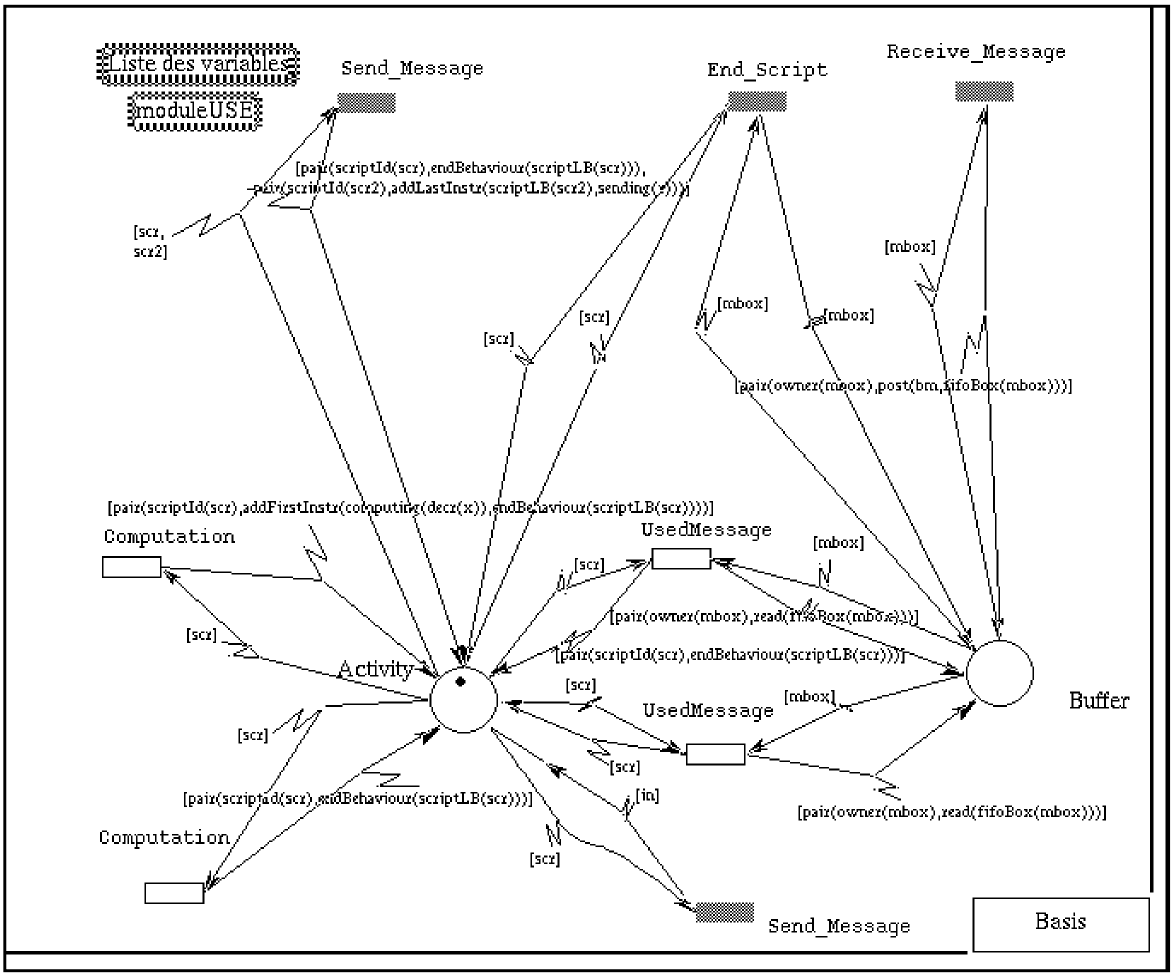


Figure 7 SANDS generated Petri net of object Basis

IV.2. Object Control

The second algorithm which is separate from the base algorithm in controlled within the control object. The latter has one place, named Color, of type PcSon. Color contains n tokens, one for each node of the system describing

the state of each node in the system (color, whether the node is active or idle, the nodes direct father in the rooted tree, a destabilizer list, the node’s identity, the list of node’s direct sons (if any)and the number of messages acknowledgments the node is waiting for). At each step of system evolution each token within Color is updated to reflect the current state of the system. The object interfaces externally through seven methods.
Events have been defined to model the behavior of the object. For example Go_Passive is synchronized through End_Script of the Basis object. Each time that a node has ended its script and has no messages to treat within its buffer (fifo), method Go_Passive updates the latter token within the place Color, namely it puts the node’s state to idle, see figure 8.

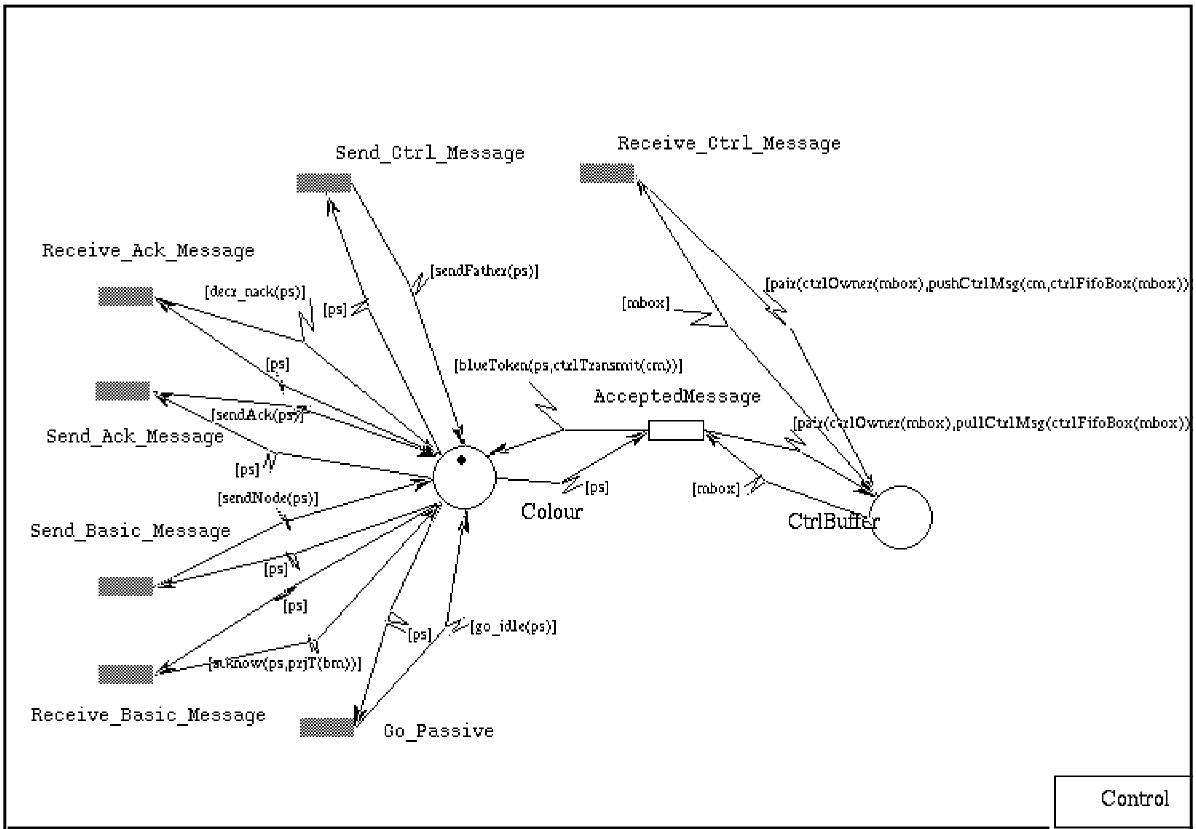


Figure 8 SANDS generated Petri net of object Control

IV.3. Object Basic_Comm

The base messages in transit within the system, namely those exchanged between nodes, are directed through a base topology defined within the place OrientedGraph of the Basic_Comm object. This object controls both Control and the Basis objects. The events defined are as follows:

1) The method Communication synchronizes with some methods of objects Basis and Control and models the behavior of rule 1 mentioned previously. The synchronization is done in parallel or simultaneously, represented graphically by the ‘II’ symbol on the links. The latter method delivers the triplet <message, transmitter, receiver>, namely of type `BasicMessage`, to the fifo buffer of node receptor, the latter residing in place Buffer. Simultaneously, methods Send_Basic_Message and Receive_Basic_Message are destined to modify the state of the transmitter node and receiver node represented in place Color, using rule 1. It is to be remembered that the calculation of the path for message delivery is done within the place OrientedGraph.

For example, if a message, such as <message, N0,N4> is to be delivered, the messages will be decomposed to <message,N0,N1> and <message,N1,N4>. The first component will be retransmitted to the buffer of token N1. In this way communication between two distant nodes is calculated. Place OrientedGraph describes a total order between the base channels.

2) Method Acknowledge is synchronized in parallel with Send_Ack and Receive_Ack methods. It delivers the acknowledgment of a basic message from node i to node j. It modifies the appropriate tokens of place Color. The acknowledgment of a basic message is done between two neighbor nodes because the path of a message is divided into elementary paths which correspond to a path between two neighbor nodes. As a result, each elementary message is acknowledged.

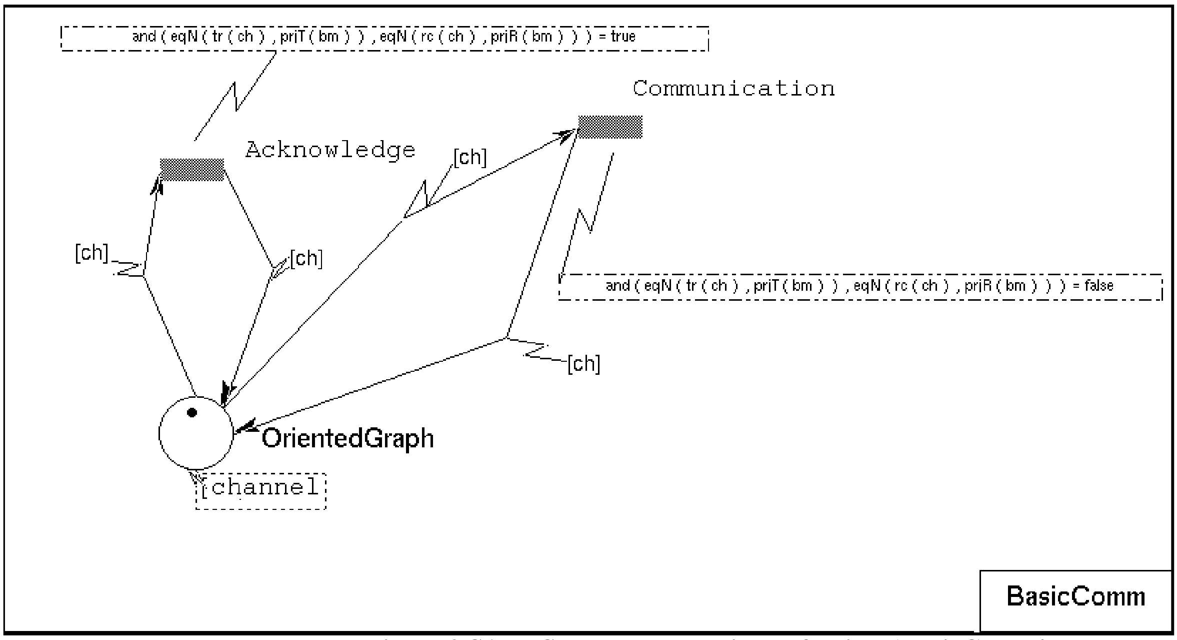


Figure 9 SANDS generated Petri net of object 'BasicComm'

IV.4. Object Control_Comm

The last object of the system modelled is similar to Basic_Comm with the difference that it describes the delivery of control messages within the rooted control tree. The place Control-Tree contains knowledge of the structure of the tree and node positions. It contains an internal marking similar to that of Basic_Comm describing immediate neighbors with pair(Ni,Nj)
tokens. Each message is delivered to the direct father node and message decomposing is no more necessary.

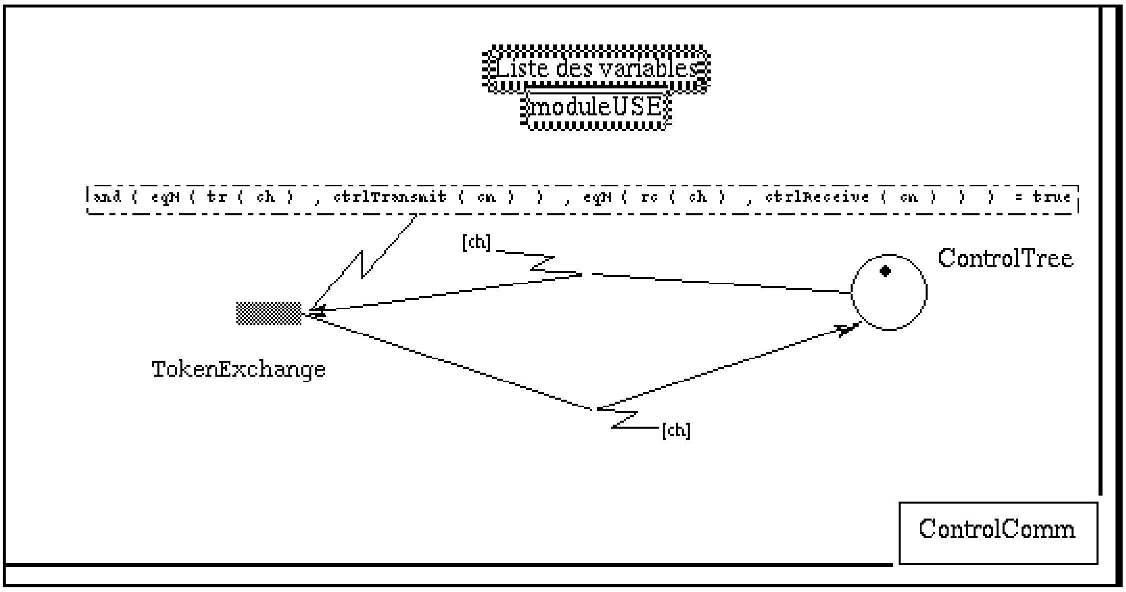


Figure 10 SANDS generated Petri net of object ControlComm

V Implementation

The system modelled through CO-OPN has been implemented on a transputer network [2].

The overall system modelled with the four objects takes in consideration all the n nodes present in the system. Places Color, Activity and Buffer all possess n tokens, of different sorts, and represent every single node in the basic algorithm. A CO-OPN object can be considered as an ADA task type and the tokens in places correspond to the instantiations of the latter type.

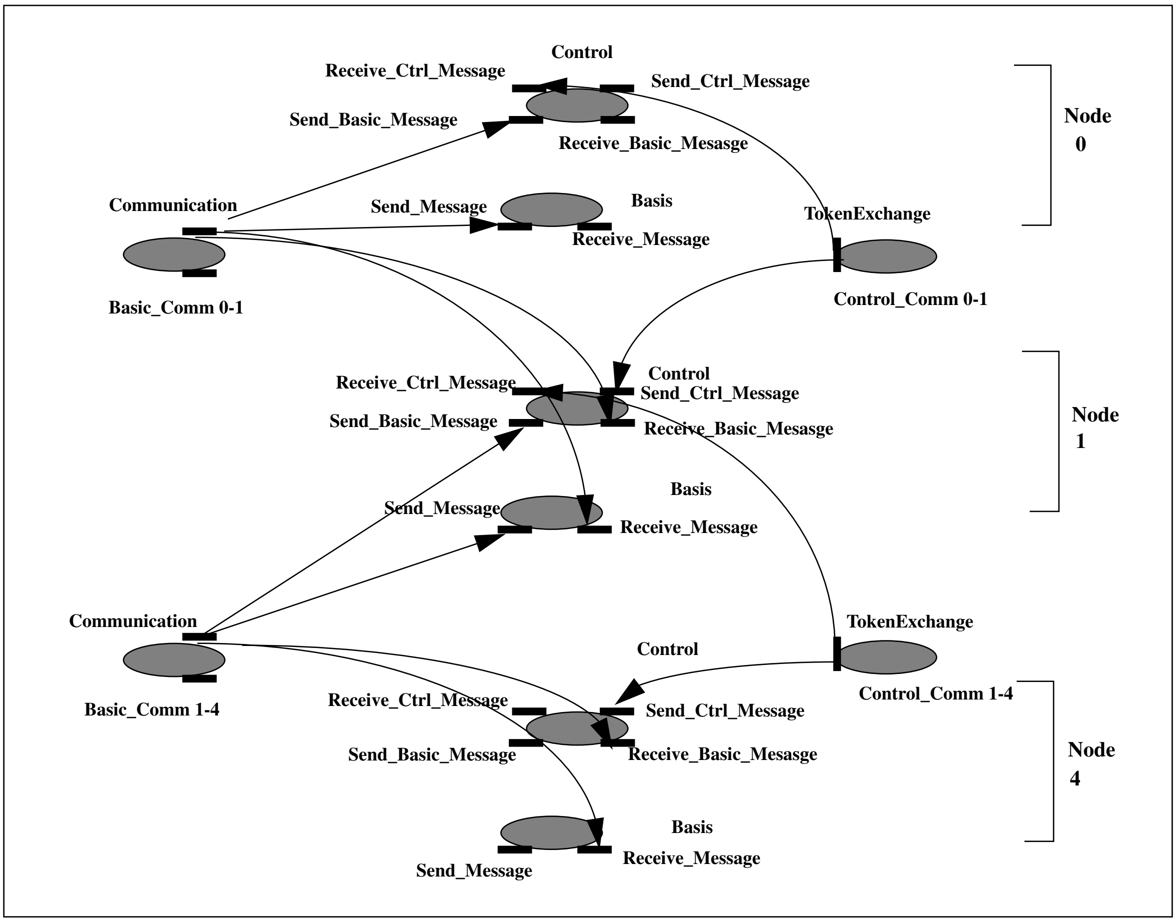


figure 11 Layer view of CO-OPN objects

The same CO-OPN object representation reduced to a single token per place would represent each process individually. In other words, reducing the place tokens to unity would allow to see each process evolve, and having n objects would represent the system globally, see figure 11.

The different objects, Control_Comm and Basic_Comm would carry out the communications between nodes and the Buffer in Basis would contain no more than a single fifo for message reception per node. Place Color would contain, as well, a single token of type PcSon representing a unique node. The distribution of the communication objects is not easy because of the complexity of the topology. In our example the base topology is a ring. Thus, there is at least one object Basic_Comm between two Basis objects (see figures 11,12). If the base topology is a DeBrayn topology the whole system becomes more complex. As a result, we can see that the general system presented within figure 3 can be seen as an n layer object specification where each place contains the state of each node in the system and places in objects possess a unique token, and not n tokens as in the general system specification.

VI Analysis of properties

The safety and liveliness properties, meaning the protocol does not produce false detections and the protocol must report termination detection in a finite time, respectively, both provide correctness of thew algorithm. W e have followed the same method for demonstrating these properties as they have been used in [1]. We have transcribed each lemma, of the correctness proving, by operators expressed within the abstract data types and we demonstrate that during CO-OPN system evolution, the properties are invariants. We present in [8] that property validation can be considered as the firing of an internal transition, within the Control object, where this property is the firing condition of this transition.

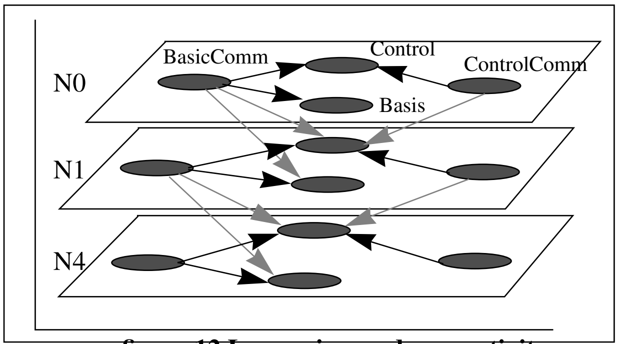


figure 12 Layer view and connectivity

VII Conclusion

In this paper we have presented aspects of the CO-OPN specification formalism. A termination detection scheme algorithm was specified with this formalism using an object oriented approach with petri nets. SANDS [3] was used to create, verify and simulate the system modelled.
The algebraic specifications provide the basis for property proving. Specification construction has aided in envisaging a particular object partitioning.

References

[1] A. Bouabdallah, M. Fouzi, ‘An ascending scheme for the distributed termination’, Proceedings of the fourth International symposium on computer and information sciences, Cesine-Izmir-Turkey, 30th October 1989 (Ed. A. Dogac, E.Gelenbe).
[2] D. Tambosco, ‘Etudes et implantation d’algorithmes de detection de terminaison sur reseau de transputers’, research report, Universite D’orsay Paris XI, 9 Sep. 1991.
[3] D. Buchs, J. Flumet, P. Racloz,’SANDS/CO-OPN’, Internal report No 71 May 1993, CUI, Université de Geneve also 14th International Conference on Application and Theory of Petri Nets, Tool presentation abstracts (Ugo Buy ed.) pp. 25-29, Chicago, USA, juin 1993..
[4] D. Buchs, J. Flumet, P. Racloz,’Producing prototypes from CO-OPN specifications’, IEEE Int. Workshop on Rapid system prototyping, Research Triangle Park, NC, june 1992.
[5] D. Buchs, N. Guelfi,’Distributed system specification using CO-OPN’, research report LRI, UPS Orsay, 1991.
[6] D. Buchs, N. Guelfi,’Open distributed programming using the object oriented specification formalism CO-OPN’, research report No 700, LRI, UPS Orsay, 1991 and also 5th International Conference on: Putting into Practice Methods and Tools for Information system design, Nantes, France september 1992, pp 261-279.
[7] D. Buchs, N. Guelfi,’ CO-OPN: A Concurrent Object Oriented Petri Net Approach’, 12th International Conference on theory and application of Petri Nets, Aahrus, 1991.
[8] D. Buchs, F. Mourlin, L. Safavi,’Distributed termination detection scheme using CO-OPN’, internal report 84, June 1993, CUI, University of Geneva
[9] N. Francez, ‘Distributed termination’, ACM transcations and programming languages and systems, VOL II No. 1 pp42-55, 1980.