CAHIERS
du
CUI

SANDS/CO-OPN

Structured Algebraic Net Development System
Concurrent Object-Oriented Petri Nets

D. Buchs J. Flumet P. Racloz
C.U.I. - Université de Genève
24 rue du Général Dufour
CH-1211 Genève, Switzerland
email: buchs , flumet, racloz@cui.unige.ch

No. 71
May 1993

DOCUMENT INTERNE

Centre Universitaire d'Informatique

Université de Genève
24, rue Général-Dufour
1211 Genève 4

SANDS/CO-OPN

Structured Algebraic Net Development System

Concurrent Object-Oriented Petri Nets

Version 2.1, May 1993

For further information please contact:

Didier Buchs
Centre Universitaire d'Informatique
4 rue du General Dufour,
1211 Geneva 4 Switzerland
E-mail: buchs@cui.unige.ch
Telephone: +(41 22) 705 76 19
fax: +(41 22) 320 29 27

Table of Contents

Preface

SANDS is a development tool based on structured algebraic nets. The environment is composed of the following features:

- Algebraic nets allowing the definition and use of structured tokens by means of algebraic specifications. Apart from reducing the size of a model these nets are accurate in describing data transformations during the evolution of the system being modelled;

- Structured nets necessary for large system developments.

The latter two components form together the language used and understood by the environment.

The development environment provides the user with an application simulation tool. This component provides immediate views of any modifications brought to the system being modelled. The environment also permits modifications to the automatically generated code through manual insertions or deletions of C or Prolog instructions.

Model semantics possess a well formalized underlying theory. Connections with other theories are numerous and offer a large range of possibilities for describing and proving properties about systems. We can mention amongst other rewriting systems, bisimulation (behaviour comparison between two system models), refinement of objects, algebraic specifications and proof of temporal logic properties used as a more abstract specification language.

The user builds and structures his model as a set of algebraic nets called 'objects' representing either instances or classes depending on the context of the modelling. These objects are related to each other by synchronization links which determine and limit the way the objects can evolve concurrently. Algebraic specifications and objects can be used in three different ways: they can be generic, 'ground' or instantiated. The underlying model, namely CO-OPN (Concurrent Object-Oriented Petri Nets), is described in more depth in: 'CO-OPN: A Concurrent Object- Oriented Petri Nets' International Conference on Application and Theory of Petri Nets, Aarhus June 1991.

SANDS has two main components: the CO-OPN graphical editor and the simulation tool. Both parts are independent in the following sense: the user can develop his model inside the graphical editor and call the simulation tool from the editor, on the other hand he can use the CO-OPN specification language to describe his model in a textual way and then use the simulation tool after having verified the validity of the specification with the compiler. Moreover these possibilities can be mixed. The user can also modify the specifications by editing the file generated by the graphical editor (written in the CO-OPN language). The graphical tool can be avoided, if desired, through hand written specifications describing the model; a specification made in the CO-OPN language can also be loaded and then modified in the graphical environment. The specifications obtained can then be simulated once the compilation and linking phases have been successfully achieved. The compiling and linking can be performed through the graphical editor. An advantage of the latter editor remains that generated code through the editor remains error free for the compilation whilst manually fed code to the editor could present syntactical errors or typing mistakes. The graphical editor sets free the user from the syntactic and grammatical rules of the CO-OPN specification language.

The algebraic specification of data can be defined outside of the graphical editor[1] environment. When this has been done the latter specifications can be imported in the graphical editor where their signatures are graphically represented. This set of algebraic specifications are used as a basis which support object specifications. There is no constraint on the way the user builds his specifications. He can adopt a top-down or a bottom-up methodology. Therefore the user can set the overall objects distribution in order to structure his model and then define the behaviour of each object. On the othe hand he can build his application on a set of objects already defined. However, these approaches can also be mixed.

The set of objects appears in a so-called principal graphical window. In this window the user defines the global organization of the objects, their external methods and the synchronization links between them. Clear and precise graphical notations and conventions enable the user to read and build these synchronization links. They can be sequential, parallel or alternate. A graphical window is associated to each object, allowing to edit its behaviour described by an algebraic net. All the rules of the CO-OPN language specification are handled graphically. The user defines graphically the places, their types, the terms on input and output arcs along with any preconditioning on the methods defined. Clarity can be maintained by avoiding information overloads. Consequently, one can choose to leave aside some graphical informations whilst, preferably, keeping them in mind. Types related to places and arcs defining terms are two such examples.

The Prototyping tool

SANDS allows model simulation. The simulation, corresponding to an animation of the graphical representation, uses the classical features of an Xview application. However, graphical features such as the move of 'tokens' is not available. Symbolic representations of tokens using algebraic terms (possibly with free variables) are used and computed with resolution techniques. Apart from the simple or structured algebraic nets, it is also possible to perform the construction and simulation of classical Petri nets. This is achieved by specifying initially the notion of a token and then informing (by an option) the compiler that the specification is related to simple nets. The compiler then automatically performs the necessary transformations. In the same way the underlying Petri net can easily be obtained and simulated through its structured algebraic specification.

The simulation tool has encapsulated the notion of visibility and availability. The user can specify the set of objects, places and transitions he wants to concentrate on. Once an item is hidden all information concerning it are masked, permitting to focus on a particular aspect of the model.

The environment provides two main methods of interaction. Initially a step by step simulation can be carried out. The engine computes the set of firable transitions together with their parameter instantiations; the latter instantiations remaining as loose as possible. The user then is asked to choose a transition in this set. Once a transition is selected the tool shows the possible parameter instantiations. The user can inspect transitions while the engine performs the computation in order to propose a new set of transitions/parameters from the output marking of the last selected transition. At any moment of the simulation, the user can choose any current marking, inspect one in particular or demand information concerning a specific transition (ex. input/output markings, related parameters etc.).

1. The graphical editor is based on GRAPHTALK, a meta-tool developed by Xerox corp.

Marking graph explorer

In the second mode of interaction the user asks questions about the non-sequential behaviour of the system in a form of a synchronization constraint among events (methods or transitions). In the same way the engine calculates the set of events/parameters checking for given constraints. Sequential, parallel or non-deterministic bound constraints may also be modified. On the other hand, the set of markings can be uniquely represented using free variables on specific domains. Sub-sets of markings are automatically computed when testing the availability of transitions and at any moment the history of a simulation can be either saved or loaded.

Algebraic specification

During simulation the user can test the behaviour of abstract data type specifications, allowing free and 'ground' variable definitions (defined as short-cuts of a term expression). One can also evaluate a specification expression containing variables. Equality between two expressions can be proved using resolution techniques; free variables contained within expressions may be instantiated if required.

Temporal logic

By the end of 1993, a tool using the branching temporal logic shall be integrated within the SANDS environment. Branching temporal logic (BTL) provides a formal framework for describing and reasoning correct behaviour of parallel systems by expressing properties related to the occurrences of events in time. The operators and modalities of the BTL can be used to express important properties such as invariance, eventuality and precedence.

Symbolic model checking

Given a temporal logic formula expressing a correctness property, one can define model checking as a technique used to verify the truth of the specified property in the structure representing the parallel system. This technique can only be applied to a system with a finite set of states. Symbolic model checking overcomes this limitation by using the notion of predicates and predicate transformers already developed in the field of temporal logics. A predicate being the entity used to represent finitely an infinite set. The effects of simple predicate transformers (functions) can be linked to the temporal logic operator semantics.

Concurrent implementation

Research is currently undertaken to determine the necessary conditioning needed in order to realize an execution of a CO-OPN specification on an MIMD machine.

Characteristics

SANDS was developed using Sun workstations running OS 4.1.3.and Openwindows ver. 3.0. The environment was developed in C, Prolog and partly in Graphtalk programming languages. Source codes of the environment can be obtained free of charge. Minimum requirements, for execution, are GraphTalk and BIMProlog run times.

Acknowledgements

Special thanks to the Parallel Computing team at the University of Geneva: Olivier Biberstein, Didier Buchs, Mathieu Buffo, Jacques Flumet, Fabrice Mourlin, Pascal Racloz and Lorens Safavi.

Chapter 1

Introduction

Project CO-OPN was undertaken in order to propose a formal specification language for the design of large parallel systems. The latter was acheived through use of a modular specification language combined with Petri Nets. More precisely, CO-OPN objects are interconnected modular entities described by algebraic nets. Thus, our formalism can be seen as a modular extension of algebraic Petri nets. Tools have been built to support this formalism and are namely a graphical editor, a compiler and a simulator for prototyping specifications. The environment produced, denoted SANDS hereafter, short for Structured Algebraic Net Development System, will be the main subject of description.
The paper is organised as follows:

- Introduction to SANDS and CO-OPN;
- Description of the language supported and the compiler;
- Description of the graphical environment;
- Description of the simulator.

The overall environment is depicted in Figure 1.

Dijkstra's philosophers and a modelling of the Alternating Bit Protocol along with other specific examples are used throughout the paper in order to illustrate different aspects of SANDS.

1.1 Formal Models

Our formal model is based on a classical object oriented approach. More precisely, the encapsulation notion is ensured by methods and abstraction is supported by synchronization expressions between objects. An object is an algebraic Petri net, with parametrized transitions as methods and the possible markings of places as states. In our terminology, the events representing the evolution of the net can be of two types: *methods* or *internal transitions*. Events modify the state of the system and symbolic links between objects express the constraint between methods or internal transitions of the objects. They are described by the combination of sequential, simultaneous, alternative or empty synchronization operators. The synchronizations are *oriented* and they link the methods or internal transitions of one object to the methods of another.

The methods of an object are activated externally whilst internal transitions are locally activated as soon as their conditions are validated. The synchronizations induce a *dependency* relation between objects which must remain orderly.

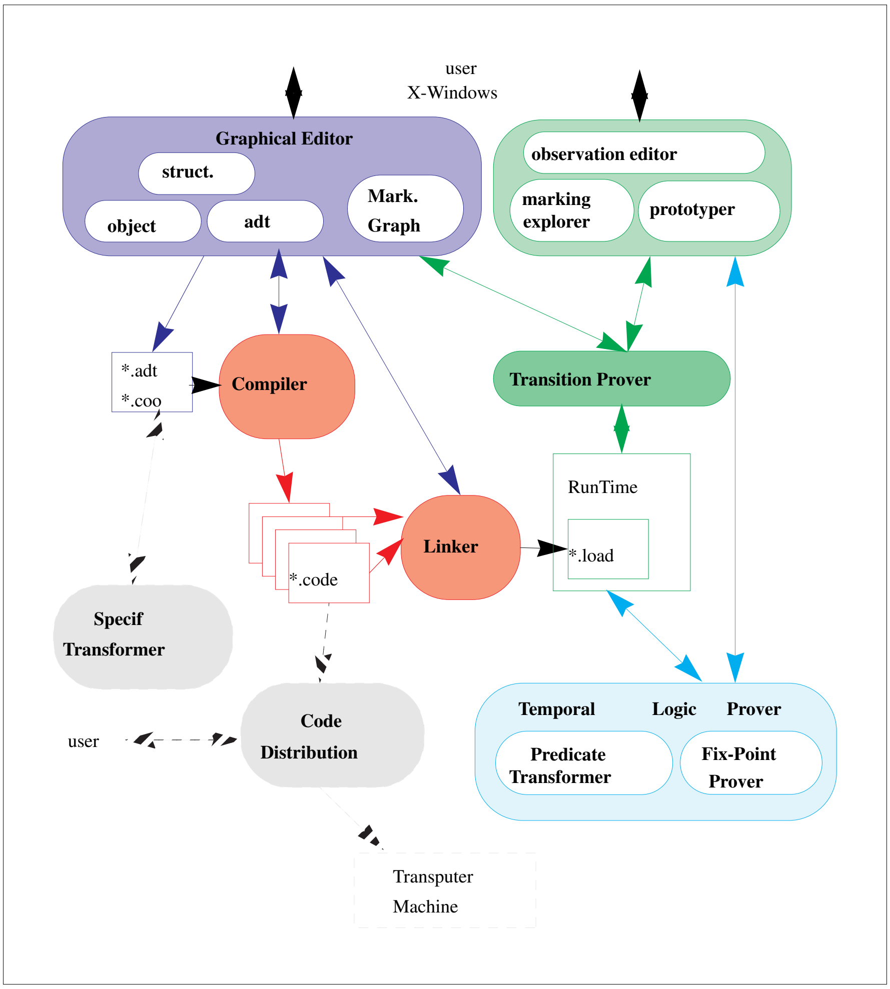


Fig. 1: Overall structure of SANDS environment

1.2 Semantics

The orderly relation between objects induces that semantics of one object are based on the semantics of its sub-objects. As a result, *hierarchy* and *modular*ity are introduced between objects. The Distributed Transition System (DTS), used to describe semantics, is then built using inference rules of the semantics for a given CO-OPN specification.

A transition of the DTS expresses concurrent evolution of system objects by using a multiset vector of concurrently firable events, each vector element corresponding to one object.

Chapter 2

The CO-OPN specification language

The description of the CO-OPN language implemented in the SANDS environment includes different aspects including the syntax and the static semantics. For the semantic aspects, references expressing its definition and properties are provided.

2.1 Lexical components

A CO-OPN specification is composed of ten different kinds of modules organised hierarchically. A module can be:
- An algebraic basic module;
- An object basic module;
- An algebraic parameter module;
- An object parameter module;
- An algebraic module enriching other algebraic modules;
- An object module enriching other object modules or algebraic modules;
- An generic algebraic module parametrized by other algebraic modules;
- An generic object module parametrized by other object modules or algebraic modules;
- An algebraic module instance of a parametrized algebraic module;
- An object module instance of a parametrized object module.

The text structure of a module is free and the file containg it will be named so as to represent the module in question, meaning that both the file and the module will carry similar names.

2.1.1 The characters

The authorized characters of the CO-OPN language are alphabetic characters (upper and lower case) and symbols such as ‘(‘, ‘)’, ‘=>’,’->’,’=’, ‘,’, ‘.’, ‘[‘, ‘]’, ’:’,’::’, ‘&’, ‘+’, ‘..’. The latter are used as separators or operators. Any other character used would induce lexical errors during compilation. Comments are enclosed between ‘%’ symbols.

2.1.2 The reserved words

The following is a list of reserved words for CO-OPN:

```
object use methods internal places where axioms with marking end
generic spec par renaming into generated by operations sorts as
( ) => -> = , . [ ] : :: & + ..
```

2.1.3 The identifiers

Identifiers are composed of a sequence of alphabetic characters (upper or lower case) and symbols.They *must* begin with an alphabetic character and should not include separators. The language remains case sensitive. For example `False` and `false` are two distinct identifiers.

2.1.4 Conventions

Grammar description rules follow the usual conventions:

- reserved words are in **bold** characters;
- alternatives are separated by: | ;
- options are surrounded by: [ ] ;
- n times entity repetition with n 0: entity* ;
- n times entity repetition with n > 0: entity+ ;
- n times entity repetition with n 0, each entity is separated with ‘’ : entity*’ ;
- n times entity repetition with n > 0, each entity is separated with ‘;’ : entity+; ;
- the ‘→’ introduces the definition of a non-terminal symbol ;
- ‘<’ and ‘>’ are used for field separations.

2.2 Algebraic basic modules

The principal objective of the CO-OPN specification language is to provide a method for expressing large system specifications. Modules are hierarchically composed allowing optimal specification construction by suppressing redundancy. These modules can be hierarchically composed and reused allowing to optimise the construction of specifications by suppressing redundancy. The definition of CO-OPN includes *enrichment*, *renaming* and *parametrizing* which are useful mechanisms for modelling real systems. The dependency relation induced by the use of basic structuring mechanisms should be a *directed acyclic graph* between modules. Relation combinations between the different possible kinds of modules follow specific rules. For instance, algebraic modules cannot enrich object modules since the expressiveness of algebraic modules are greater than that of object modules. As a result the leaves of such a graph represent the algebraic basic modules.

The major parts of a basic specification are *signature* and *property* declarations through *axioms*. A signature declaration is composed of a *sorts list*, a *generator* list and a list of other operations. An axiom declaration is composed of two parts: the axioms and the variable declarations used for writing specification terms:

```
algspec →  headeradt
           [renamingdescr]
           [usedescr]
           [sortsdescr]
           [generatorsdescr]
           [operationsdescr]
           [axiomsdescr]
           [vardescr]
           end
```

2.2.1 The signature

```
sortsdescr → sorts Ident+, ;
generatorsdescr → generated by <OpDef ;>+
operationsdescr → operations <OpDef ;>+
OpDef → Ident : Ident* -> Ident
```

The signature is composed of at least one of the following elements: a sort declaration, a generator definition and a definition of operations. The specification models are always finitely generated with respect to particular families of operators, namely the generators. The sort declaration must be followed by the corresponding generator definitions. A sort, represented by an identifier, cannot be redefined within the specification modules concerned.

Example of a signature:

```
sorts bool;

generated by

true :  -> bool;
false :  -> bool;

operations

and,or : bool , bool -> bool;
not: bool        -> bool;
equal: bool , bool -> bool;
```

This signature defines a sort `bool` generated by two constants `true` and `false`. The binary operations `and`, `or` and `equal` and the unary operation `not` are defined within the signature.

2.2.2 The terms

Operations and generators allow term construction. A term is recursively defined as:

- a constant,
- a variable,
- an operation or a generator having terms as parameters.

A variable is defined as a constant:

```
vardescr → where <VarDef ;>+
VarDef → Ident+, : Ident
```

An example of a variable declaration could be written as:

```
where
x,y:bool;
```

An example of a term declaration could be written as:

```
and(or(x,not(y)),true) not(or(true,y))
```

In cases where ambiguous definitions may occur it may be necessary to precise the module of origin of an identifier by using the notation `module_name.operator`.
Rule of the module name prefix:

```
Ident → ident | ident . ident
```

Terms are strongly typed. Given an operation `f: s1,s2,...,sn -> s` and `t1` being a term of sort `s1`, `t2` being a term of sort `s2` ... and `tn` being a term of sort `sn` then `f(t1,t2,...,tn)` is a term of sort `s`.

2.2.3 The axioms

Axioms are used to express properties of the operators of a specification and are build from the terms. Only conditionally positive axioms are allowed, the operations are total and each axiom is labelled uniquely.

```
axiomsdescr → axioms < AxiomsAdtDef ; >+
AxiomAdtDef → Ident : Atom*& => Atom
Atom → Term = Term | ( Term = Term )
```

Example:

```
axioms

aand1: => and(true,BTA) =  BTA;
aand2: => and(false,BTA) =  false;
aor1: => or(true,BTA) =  true;
aor2: => or(false,BTA) = BTA;
anot1: => not(true) = false;
anot2: => not(false) = true;
aequal1: => equal(true,true) = true;
aequal2: => equal(true,false) = false;
aequal3: => equal(false,true) = false;
aequal4: => equal(false,false) = true;


where
 BTA : bool;
```

In this example the axioms are simply equations.

2.2.4 Syntax of a basic module

A basic module is composed of a signature and axiom declarations are followed by variable definitions.

```
headeradt → spec Ident ;
algspec →  headeradt
            [renamingdescr]
            [usedescr]
            [sortsdescr]
            [generatorsdescr]
            [operationsdescr]
            [axiomsdescr]
            [vardescr]
            end
```

A specification module is identified by its name specified after the keyword `spec`.

Example:

```
spec Nat;

sorts nat;
```

```
generated by
zero   :           -> nat;
succ   : nat       -> nat;

operations
add     : nat , nat -> nat;  % addition          %
sub    : nat , nat -> nat;  % substraction      %
mul      : nat , nat -> nat;  % multiplication     %
p2       : nat       -> nat;  % square %

axioms

aadd0: => add(zero, X) = X;
aadd1: => add(succ(X),Y) = succ(add(X, Y));

asub0: => sub(zero,zero) = zero;
asub1: => sub(succ(Y), succ(X)) = sub(Y,X);
asub2: => sub(zero,succ(X)) = zero;
asub3: => sub(succ(Y), zero) = succ(Y);

amul0: => mul(zero,X) = zero;
amul1: => mul(succ(X),Y) = add(mul(X,Y),Y);

ap20: => p2(zero) = succ(zero) ;
ap21: => p2(succ(X)) = mul(succ(succ(zero)),p2(X)) ;

where
   X,Y,Z:nat;
end
```

2.3 Enrichment of algebraic modules

The most important structuring primitive of CO-OPN is enrichment. It permits to add new elements (sorts and operators) and properties to already defined algebraic modules. From the point of view of the module defining the enrichment, the enriched modules are called *required modules.*

When a specification module is required its model cannot be changed. The addition of new operations must not change the existing properties. This fundamental rule follows from the hierarchical constraint described earlier. In particular, if it is possible to declare new operations with a sort of a required module as its co-domain, new values must not be added to these sorts. Moreover, the generators and their sorts must be defined in the same module.

By default, the signature of a required module is completely visible from the modules that require it directly (the visibility is not transitive unlike the enrichment). The choice made in the PLUSS language is, however, different. Our approach does not permit the signature to grow uncontrolled during the enrichment process. To render this control possible it is necessary to build a list of visible modules, even if these modules are indirectly enriched.

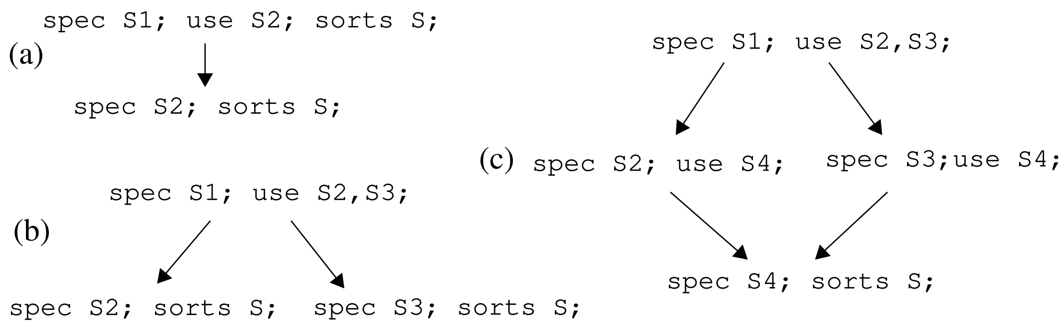
2.3.1 Syntax of the algebraic module enrichment

usedescr $\rightarrow$ **use** Ident$^{+,}$;

The module name referenced following the reserved word `use` corresponds to the modules' name defined immediately after the keyword `spec`.

2.3.2 Static semantics of the enrichment

In a CO-OPN specification, an element cannot be defined many times (a). For instance, it is impossible to define a sort already defined in another required module(b) or to require two specification modules in which the same sort is defined(b). Nevertheless, it is possible to indirectly enrich modules using different ways (c).



In (c), the enrichment of S4 does not allow to have sort S visible in S1, it may be necessary to explicitly write S4 in the `use` list of S1.

`Vis(S)`={sorts, generators and operations defined in module S}
`Use(S)`={modules required by S}

`Util(S)`={sorts, generators and operations usable within the terms of module S}= $\bigcup_{\mathbf{MOD} \in \mathbf{Use}(\mathbf{S})} \mathbf{Vis}(\mathbf{MOD})$

Example:

```
spec Nat;

use Bool;

sorts nat;

generated by
zero  :           -> nat;
succ  : nat       -> nat;

operations
add         : nat , nat -> nat;  % addition           %
sub         : nat , nat -> nat;  % soustraction       %
mul         : nat , nat -> nat;  % multiplication     %
equal       : nat , nat -> bool; % test of equality  %
le          : nat , nat -> bool; % lower or equal%
lt          : nat , nat -> bool; % lower       %
ge          : nat , nat -> bool; % greater or equal%
gt          : nat , nat -> bool; % greater        %
even        : nat       -> bool; % parity         %
max         : nat , nat -> nat;  % maximum %

axioms
a0: => add(zero, X) = X;
```

```
a1: => add(succ(X),Y) = succ(add(X, Y));

a0: => sub(zero,zero) = zero;
a1: => sub(succ(Y), succ(X)) = sub(Y,X);
a2: => sub(zero,succ(X)) = zero;
a3: => sub(succ(Y), zero) = succ(Y);

a0: => mul(zero,X) = zero;
a1: => mul(succ(X),Y) = add(mul(X,Y),Y);

a0: => equal(succ(X),succ(Y)) = equal(X,Y);
a1: => equal(zero,succ(X)) = false;
a2: => equal(succ(X),zero) = false;
a3: => equal(zero,zero) = true;

a0: => le(X,Y) = or(equal(X,Y),lt(X,Y));

a0: => lt(succ(X),succ(Y)) = lt(X,Y);
a1: => lt(zero,succ(X)) = true;
a2: => lt(succ(X),zero) = false;
a3: => lt(zero,zero) = false;

a0: => ge(X,Y) = le(Y,X);

a1: => gt(X,Y) = lt(Y,X);

a0: => even(zero) = true;
a1: => even(succ(X)) = not(even(X));

a0: gt(X,Y) = true => max(X,Y) = X ;
a1: gt(X,Y) = false => max(X,Y) = Y ;

where
   X,Y,Z:nat;
end
```

2.3.3 Renaming

It is possible to rename certain elements of a specification to avoid possible confusions with other identifiers:

```
renamingdescr → renaming <RenameDef ;>+
RenameDef → Ident into Ident
```

A renaming of a module S is a set `Ren(S)` of couples `(old_name,new_name)`. These couples define a renaming function in the following way:

- `rename(id,Ren(S))=id if (id,idn) ∉ Ren(S)`
- `rename(id,Ren(S))=idn if (id,idn) ∈ Ren(S)`

The visibility of the identifier is then:
`VisR(S)`={sorts, generators and operations defined in module S after renaming}=
`{id | id=rename(id',Ren(S)) such that id' ∈ Vis(S)}`

The available identifiers are then: $\mathtt{Util(S)} = \bigcup_{\mathbf{MOD} \in \mathbf{Use}(\mathbf{S})} \mathbf{VisR}(\mathbf{MOD})$

2.4 Generic algebraic modules

It is inefficient to build specifications from scratch. The enrichment primitive allows to incrementally build specifications. In the same way, it is less expensive to generalize a specifications and to instantiate it many times instead of writing numerous quasi equivalent copies of the same specification. For instance, the specification of the stack of integers must be completely rewritten even if a specification of the stack of boolean is known. In CO-OPN the generic modules are allowed for this purpose.

2.4.1 Generic algebraic specification modules

The syntax of the generic algebraic specification module and the parameter algebraic specification module are defined by the following rules:

```
headeradt → [generic | par] spec
            Ident [ParameterBloc] [as Ident [ParameterBloc]] ;
ParameterBloc → ( Ident+, )

algspec →   headeradt
            [renamingdescr]
            [usedescr]
            [sortsdescr]
            [generatorsdescr]
            [operationsdescr]
            [axiomsdescr]
            [vardescr]
            end
```

Example of a generic specification, a Lifo:

```
generic spec Lifo( Elem);
 use Bool, Elem;
 sorts lifo;
 generated by
 iempty: -> lifo;
 push: elem, lifo -> lifo;
 operations
 isEmpty: lifo -> bool;
 pop:  lifo -> lifo;
 top: lifo -> elem;
 ERREUR:-> lifo;
 axioms
 a0: => isEmpty( empty) = true;
 a1: => isEmpty( push( e, b)), false)).
 a0: => pop( empty) = ERREUR;
 a1: => pop( push( e, b)) = b;
 a0: => top( empty) = ERREUR;
 a1: => top( push( e, b)) = e;
 where b: lifo;
       e: elem;
end
```

The parameter specification must be enumerated in the field `use`.

2.4.2 Syntax of formal parameters

The syntax of the formal parameters is a simplification of that of specifications. A formal parameter has to be instantiated by a specification, it is usually limited and it has often no operations. The algebraic modules of a formal parameter do not have to be finitely generated, there exist no declarations of generators.

```
par spec Elem;
 use Bool;
 sorts elem;
 operations
 equal: elem, elem -> bool;
 ERREUR : -> elem;
end
```

2.4.3 Instantiations

The replacement of formal parameters by specification modules denoted `spec` are called *instantiation*. If only a sub-set of the parameters are replaced, the instantiation is called *partial*. The replacement of a `par` by an other `par` is then called *formal* instantiation. Until all formal parameters have been replaced by concrete ones, a specification is known as generic specification (`generic spec`).

The instantiation of `par` has to preserve semantics. The models associated to the actual parameters must have the properties required by the formal parameters. Moreover, the models of the specification `spec` must not be changed. From the `spec` point of view, an instantiation is equivalent to an enrichment.

```
spec Lifonat as Lifo( Nat);
 renaming lifo into lifonat;
 use Nat, Bool, Lifo, Elem;
operations
 equal: lifonat, lifonat -> bool;
 axioms
 a0: => equal( empty, empty) = true;
 a1: => equal( empty, push( e2, b2)) = false;
 a2: => equal( push( e1, b1), empty) = false;
 a3: (equal( e1, e2) = true) =>
   ( equal( push( e1, b1), push( e2, b2)) = equal( b1, b2));
 a4: ( equal( e1, e2) = false) =>
         ( equal( push( e1, b1), push( e2, b2)) = false);
 where b1, b2: lifonat;
  e1, e2: nat;
end
```

```
headeradt → spec
            Ident [ParameterBloc] [as Ident [ParameterBloc]] ;
ParameterBloc → ( Ident+, )
```

When a `generic spec` has more than one `par`, they must be separated by commas. The *morphism* between `pars` and the instantiation `spec` is set to identity, renaming allows to modify the identifier of the instantiated `spec`.

2.4.4 Static semantics of the instantiation

The static semantics consider the `par` module as the required modules. The instantiation is the substitution in the signature of the `generic spec` of the elements of the `par` by those of the `spec`; the `par` axioms are removed and those of `the spec` are added. The operations of the `generic spec`, in the profile of which a sort of the `par` appears, are as a result renamed.

2.5 Basic objects

A basic object is composed of three different notions: the objects' signature, the state variables and the axiom declarations defining its behaviour. A signature describes the set of methods and internal transition; the state variables along with the places are a list of typed identifiers; an axiom declaration is followed by a list of variables.

```
objectspec →  headerobject
              [renamingdescr]
              [usedescr]
              [placesdescr]
              [methodsdescr]
              [intdescr]
              [axiomsobjectdescr]
              [markingdescr]
              [vardescr]
              end
```

It is to be noted that the declaration and renaming of variables follows the same description of algebraic modules.

2.5.1 The signature

```
methodsdescr → methods <MethDef ;>+
MethDef → Ident : Ident*
```

The signature is composed of method declarations and the *profile* of a method is defined by a list of sorts.

Example of a signature:

```
object Buffer_mem;

use Lifonat,Nat,Bool;

methods put:nat; get:nat;
```

This signature defines an object `Buffer_mem` accessible through two methods `put` and `get` of sort `nat`.

2.5.2 The terms

The algebraic specifications enriched in an object specification with a `use` clause define the set of algebraic terms in the same way as described earlier. Three other classes of terms may be used within objects. They are composed of event terms, rendez-vous (or synchronization) and the multi-set terms.

Event terms

These terms are build strongly typed except that the head of a term must be an events' name, meaning either a method or an internal transition. Note that event names appear only at a head position.

Example:

```
put(push(succ(zero),empty))
```

Synchronization terms

These terms are build over event terms wit three different basic synchronization operators. These are namely the ‘&’ for *simultaneous*, ‘+’ for *alternative* and the ‘..’ operator for *sequence* or sequential synchronization.

```
SynchroTerm → TermSync & SynchroExpr | TermSync + SynchroExpr |
TermSync .. SynchroExpr | SynchroExpr & TermSync | SynchroExpr +
TermSync | SynchroExpr .. TermSync | TermSync
SynchroExpr → ( SynchroTerm )
TermSync → Term
```

Example of a synchronization term:

```
(put(push(succ(zero),empty))..get(X))|get(zero)
```

Multiset terms

In order to be able to model a collection of data, a multiset sort is implicitly defined for each existing sort:

- `nat`: name of a sort;
- `[nat]` : new multiset sort of `nat`.

The corresponding terms are also predefined. The multiset terms of sort `nat` being for instance:

```
[zero,succ(zero),succ(X),zero]
```

2.5.3 State variables

Definition of places

The state of an object is described by the places of an algebraic net. A place contains values of a sort defined in an algebraic module.

```
placesdescr → places <PlacesDef ;>+
PlacesDef → Ident+, : Ident
```

Example:

```
places buffer:[lifonat];
```

Places in axioms

The terms describing the precondition and postcondition of events are defined by a list of place modifiers. A place modifier is composed of a tuple: the place name and the multiset term modifying the places.

```
PlacesTermList → PlaceModif*,
PlaceModif → Ident ( [ Term*, ] ) | Ident ( Term )
```

Example of a term:

```
places value:[nat]

...
value([zero,succ(zero),succ(X),zero])
```

Initial Marking

The initial state of an algebraic net is defined by a list of place modifiers (usually without any variables).

markingdescr → **marking** PlacesTermList ;

Example:

```
places buffer:[lifonat];
...
marking buffer([empty]);
```

2.5.4 The internal transitions

The internal transitions are used to represent events depending only on the state of the object and not on any external events. An internal transition is declared after the keyword `internal`. An internal transition cannot be used in a synchronization expression and they cannot be parametrized. Nevertheless, an internal transition can call other methods in a synchronization expression.

intdescr → **internal** Ident$^{+,}$;

Example:

```
internal flush;
```

2.5.5 The axioms

The axioms are used to describe the properties of the events of a specification, built over the possible terms. The authorized axioms are positive conditional axioms with precondition and postcondition on the state variables and a synchronization part used to enrich other objects. Each axiom being uniquely identified in order to designate it during the compilation process:

axiomidentifier: methodnamewithparameter `with` synchronizationexpression `::`
condition =>
precondition -> postcondition;

axiomsobjectdescr → **axioms** <AxiomsObjectsDef ;>$^+$
AxiomsObjectsDef →
Ident : Term [**with** SynchroTerm] ::
Atom$^{*\&}$ **=>** PlacesTermList **->** PlacesTermList ;

Example:

```
axioms
a0:put(m):: => buffer([b]) -> buffer([push(m,b)]);

a1:get(top(b))::(not(isEmpty(b))=true)=>
 buffer([b]) -> buffer([pop(b)];
where b:lifonat; m:nat;
```

In this example the first axiom has no condition and both axioms have no synchronization.

2.5.6 Syntax of a basic object module

A basic object module is composed of a signature, an axiom declaration and a variable declaration:

```
objectspec →  headerobject
              [renamingdescr]
              [usedescr]
              [placesdescr]
              [methodsdescr]
              [intdescr]
              [axiomsobjectdescr]
              [markingdescr]
              [vardescr]
              end

headerobject :object Ident;
```

A specification object module is identified by its name following the keyword `object`. Any reference to this module will be made with this latter identifier.

Example:

```
object Buffer_mem;

use Lifonat,Nat,Bool;

places buffer:[lifonat];

methods put:nat; get:nat;

axioms
a0:put(m):: => buffer([b]) -> buffer([push(m,b)]);

a1:get(top(b))::(not(isEmpty(b))=true)=>
 buffer([b]) -> buffer([pop(b)];
where b:lifonat; m:nat;

init_marking buffer([empty]);

end
```

2.6 Enrichment of objects

Objects can be used in the same manner as the algebraic modules by using the keyword `use`. Methods of the enriched modules are used by including them in the synchronization expressions. A synchronization expression corresponds to a condition to the firing of a transition.

Example:

```
object Consumer;
use Buffer_mem,Nat;
methods receive;
```

```
axioms
a1:receive with get(n):: => ->;
where n:nat;
end

object Producer;
use Nat, Buffer_mem;
places counter:[nat];
methods send:nat;
axioms
a1:send( m) with put( m) ::
counter([ n]) -> counter([succ( n)]);
 marking counter([ zero]);
where m,n:nat;
end
```

2.7 Generic object modules

The generic object modules are defined in the same way as the generic algebraic modules. A generic object module can be parametrized by parameter algebraic modules or parameter object module.

2.7.1 Generic object modules

The syntax of the generic object and the parameter object are defined by the following rules:

```
headerobject → [generic | par] object
            Ident [ParameterBloc] [as Ident [ParameterBloc]] ;
ParameterBloc → ( Ident+, )
objectspec →  headerobject
               [renamingdescr]
               [usedescr]
               [placesdescr]
               [methodsdescr]
               [intdescr]
               [axiomsobjectdescr]
               [markingdescr]
               [vardescr]
               end
```

Example of a generic object, a buffer:

```
generic object Buffer_mem(Struct);

use Struct,Nat,Bool;

places buffer:[lifo];

methods put:elem; get:elem;

axioms
a0:put(m):: => buffer([b]) -> buffer([push(m,b)]);

a1:get(top(b))::(not(isEmpty(b))=true) =>
 buffer([b]) -> buffer([pop(b)];
```

```
where b:lifo; m:elem;

init_marking buffer([empty]);

end
```

The parameter specification must be enumerated in the field `use`.

2.7.2 Syntax formal parameter objects

The syntax of the formal parameters are a simplification of those of the objects. A formal parameter has to be instantiated by an object and is usually limited having no axioms and sometimes no methods:

```
par object Time;
use Bool;
methods
 alarm:bool;
end
```

2.7.3 Instantiations

The replacement of formal parameters by `object elements` or algebraic modules `spec` is denoted *instantiation*. If only a sub-set of the parameters are replaced, the instantiation is called *partial*. The replacement of a `par` by another `par` is named *formal* instantiation. Until all formal parameters have been replaced by concrete ones, a specification remains a generic specification (`generic spec`).

The instantiation of `par` has to preserve its semantics. The models associated to the actual parameters must have the properties required by the formal parameters. Moreover the models of the specification `spec` must not be changed. From the `spec` point of view, an instantiation is equivalent to an enrichment. For the par object only the profile of the methods must be preserved.

```
object Buffer_mem_nat as Buffer_mem(Lifonat);

use Lifonat,Nat,Bool;
end

headerobject → object
            Ident [ParameterBloc] [as Ident [ParameterBloc]] ;
ParameterBloc → ( Ident+, )
```

When a `generic spec` has more than one `par` the latter must be separated by commas. The morphism between `par` and the instantiation `spec` or `object` being set to identity and renaming allows modification of the instantiated `spec` identifiers.

2.7.4 Static semantics of the instantiation

The static semantics consider the `par` module as required modules. The instantiation is the substitution in the signature of the `generic object` of the elements of the `par` by those of the `spec` or `object`; the `par` axioms are removed and those of `spec` or `object` are added. The methods of the `generic object` in the profile of which a sort of the `par` appears are renamed.

Chapter 3

Graphical Representations

3.1 Introduction

Within this section we will focus attention on the CO-OPN graphical notations used by the graphical editor. To do so, we will initially present notations representing objects and algebraic specifications. We will then pursue with the presentation of Petri net graphical representations and conclude by demonstrating the graphical representation of synchronization expressions which link different objects together.

3.2 Algebraic basic modules

Figure 2 depicts an example of the representation used for an algebraic basic module, denoted 'MADT1' in the example, within the graphical environment.

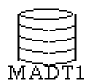


Fig. 2: Basic algebraic specification

3.2.1 The signature

The graphical description allows representations of signatures; a signature is composed of at least one of the following elements: a sort declaration, a generator definition or a definition of operations. Figure 3 depicts the representation of the boolean specification denoted as 'Bool'. Sort 'bool' is represented under the module name, the generators are also listed with their input output sorts. Operation profiles are also listed.

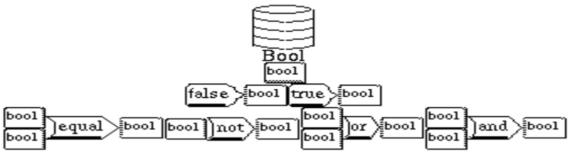


Fig. 3: Specification of 'Bool'

3.3 Enrichment of algebraic modules

The most important structuring primitive of CO-OPN is enrichment. It permits to add new elements, including sorts and operators, as well as properties to already defined algebraic modules. Enriched modules are listed in the graphical representation of a module.
Figure 4 depicts an example where module 'Nat', representing the natural numbers, enriches the 'Bool' module.

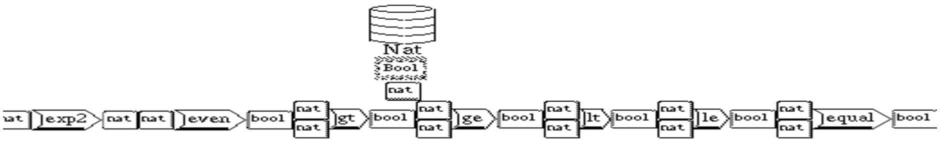


Fig. 4: Specification of 'Bool' enrichment by module 'Nat'

3.4 Generic algebraic modules

The graphical representation of the generic algebraic specification is based on the representation of the basic objects. The arrow indicates that information feed are required.

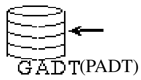


Fig. 5: Generic specification module

3.4.1 Representation of formal parameters

The graphical representation of the parameter algebraic specification module is also based on the basic specification modules (see Figure 2) and an arrow representing that information is provided to an generic module.

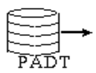


Fig. 6: Parameter specification module

3.4.2 Instantiations

In this case no particular representations are provided.

3.5 Basic objects

A basic object named 'o1' is represented as in Figure 7.

Fig. 7: Basic object

3.5.1 The signature

The signature of an object is composed of method declarations represented as black boxes. The *profile* of a method is defined by a list of sorts.

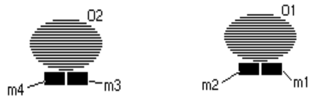


Fig. 8: Basic object with methods

Event terms

These terms are build strongly typed and the head of a term must be composed of an event name.

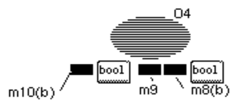


Fig. 9: Basic object with parametrized methods

Synchronization terms

These terms are build over event terms with three different basic synchronization operators each having their own graphical representations. They are namely the '&' operator denoting *simultaneous*, the '+' operator representing *alternative* and the '..' operator for *sequence* or sequential synchronization. The labels on the arcs represent the call of method defined in the synchronization expression. Terms are composed of variables and operators derived from inside the calling object.

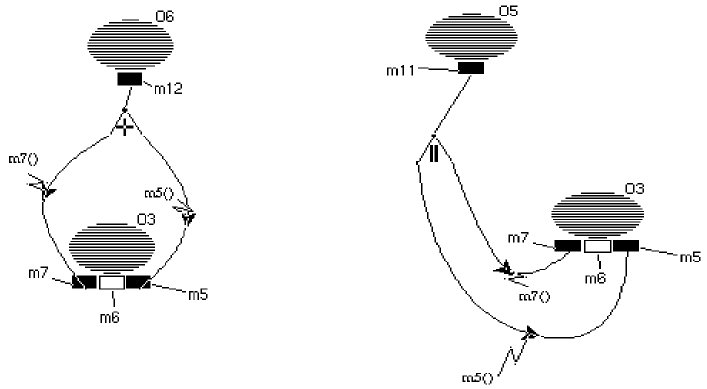


Fig. 10: Basic object with synchronizations (alternative and simultaneous)

A synchronization term with more than one synchronization operator is represented in Figure 11.

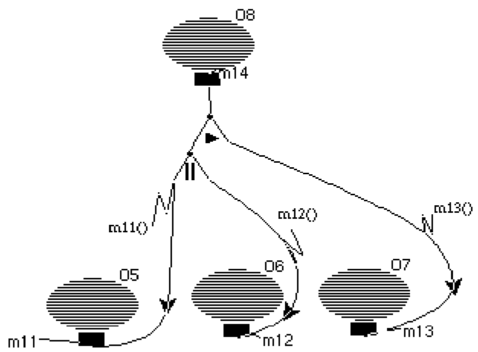


Fig. 11: Basic object with synchronization term `m14 with (m12 & (m11 ) ) .. m13`

A complex structure of multi-hierarchical level of synchronization expressions on objects is represented in Figure 12.

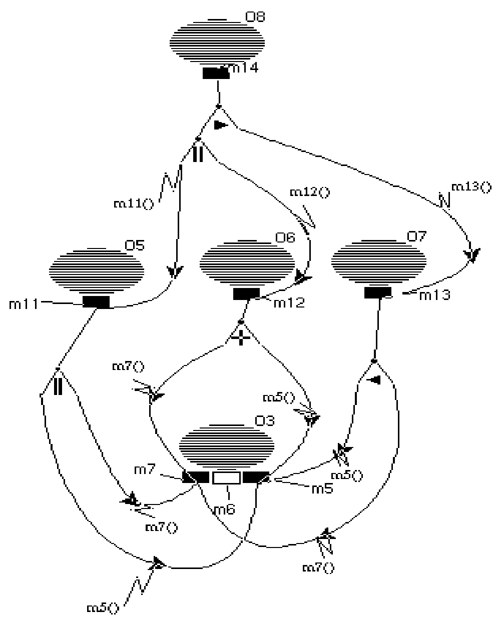


Fig. 12: Hierarchy of synchronization

3.5.2 State variables

Definition of places

The state of an object is described by its algebraic net places. The latter are represented graphically by a circle. A place contains values of a sort represented by the label attached to it. Figure 13 represents places of sort 'bool'.

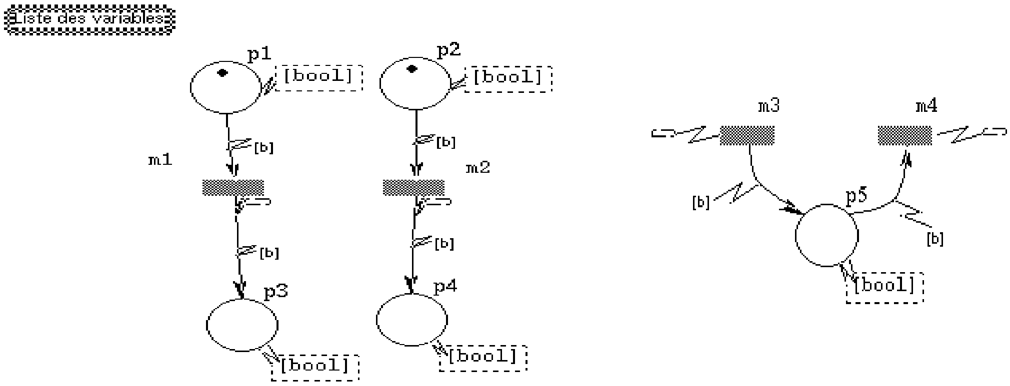


Fig. 13: Different causality relations

Initial Marking

The initial state of an algebraic net is represented with an initial marking. A black point inside a place represents a marking, the value of which is accessible through the editor.

3.5.3 The internal transitions

The internal transitions are used to represent events depending only on the state of the object and not on any external events. An internal transition is represented by white boxes with an arrow inside as depicted in Figure 14.

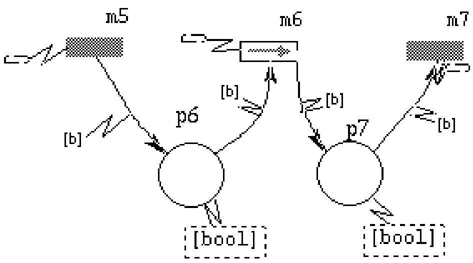


Fig. 14: Internal transitions

3.5.4 The axioms

Axioms are used to describe the properties of the events of a specification. The conditions are attached to the transitions or events. Arcs linking places to events, and events to places, are labelled by multisets representing the pre and post conditions.

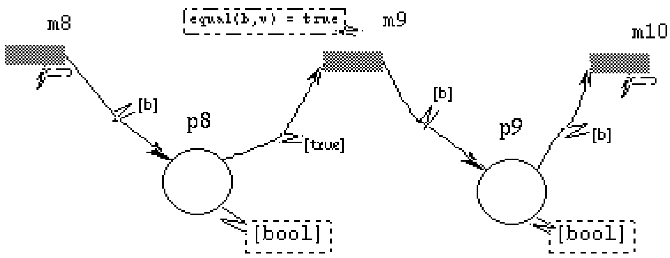


Fig. 15: Condition and terms on nets

3.6 Enrichment of objects

Objects can be used in the same manner as the algebraic modules by using the keyword `use`. No explicit representations are currently provided.

3.7 Generic object modules

The generic object modules are defined in the same way as the generic algebraic modules. A generic object module can be parametrized by parameter algebraic modules or parameter object modules.

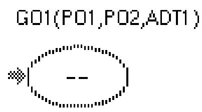


Fig. 16: Generic object module

3.7.1 Syntax formal parameter objects

A formal parameter is usually limited to having no axioms and sometimes no methods.

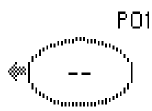


Fig. 17: Parameter object module

3.7.2 Instantiations

The replacement of formal parameters by `'object'` is represented by white objects with the name of the generic objects inside it.

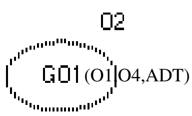


Fig. 18: Instantiated object module

Chapter 4

The graphical editor

4.1 Introduction to the graphical tool

The two graphical windows allow the development of algebraic specifications and CO-OPN objects of the system modelled. Currently algebraic specifications are developed through a text editor and restrictions are imposed over the usage of the environment. The latter are as follows:

- The model's acquires a unique name throughout modelling;

- The 'Inspect' command should be avoided. The latter inflicts over modelling coherency and, being attached to the Graphtalk system directly, it could cause anomalies in the model's behaviour.

4.2 Algebraic specification window

CO-OPN objects are built over algebraic specification present in the graphical window. As a result, it is necessary to include, within the latter window, all necessary algebraic specifications needed for the system being modelled. Algebraic specifications can be edited, imported to the graphical environment at which point they are compiled and linked automatically. The example below shows the algebraic specification window for model 'Philos'.

Figure 19 shows the subitem 'ReadAlgFile' being activated inorder to import a textual specification. If the file exists, it is imported, compiled and represented graphically.

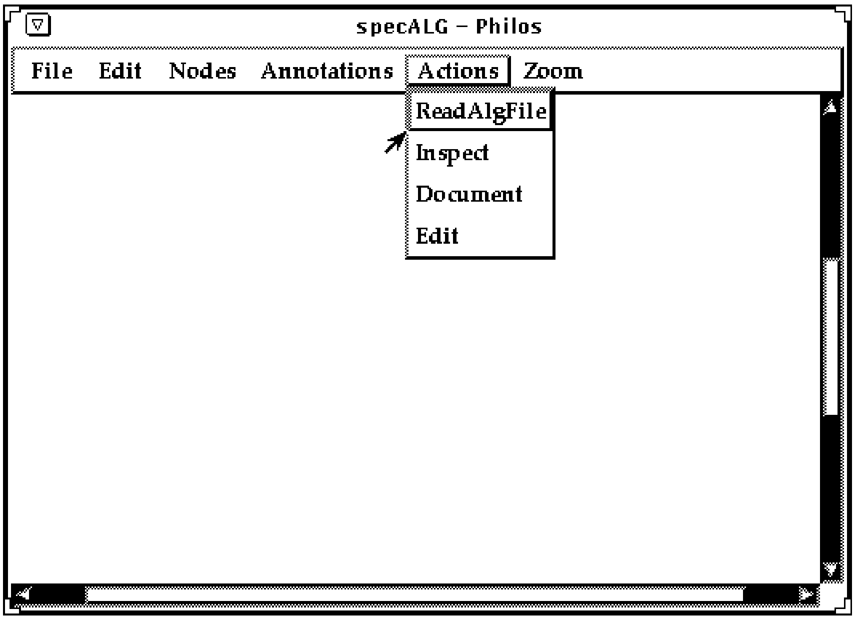


Fig. 19: Importing of an algebraic specification

It is necessary to compile the files of algebraic specifications before compiling the CO-OPN objects. The user clicks twice on the object which represents the file of the specification and selects 'Compilation'.

.

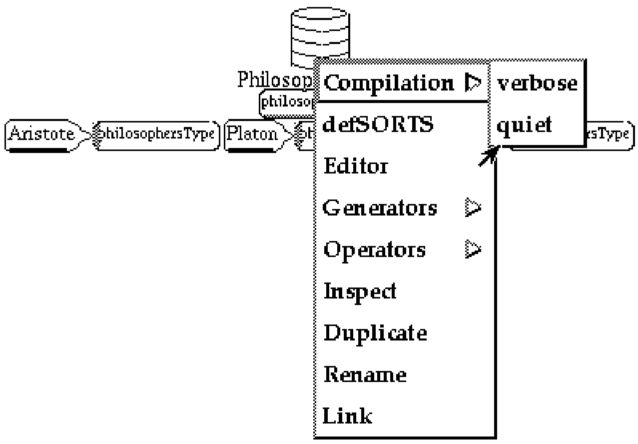


Fig. 20: Menu on the graphical representation of an algebraic specification.

Remark

The `'Compilation'` command parses the text file and generates the associated code. But it does not create a graphical representation. If changes were made previously the file has to be read again through the 'ReadAlgFile' command.

4.3 CO-OPN objects window

4.3.1 CO-OPN objects

A CO-OPN object is created by selecting the submenu `'object'` in the `'Nodes'` menu (see Figure 21).

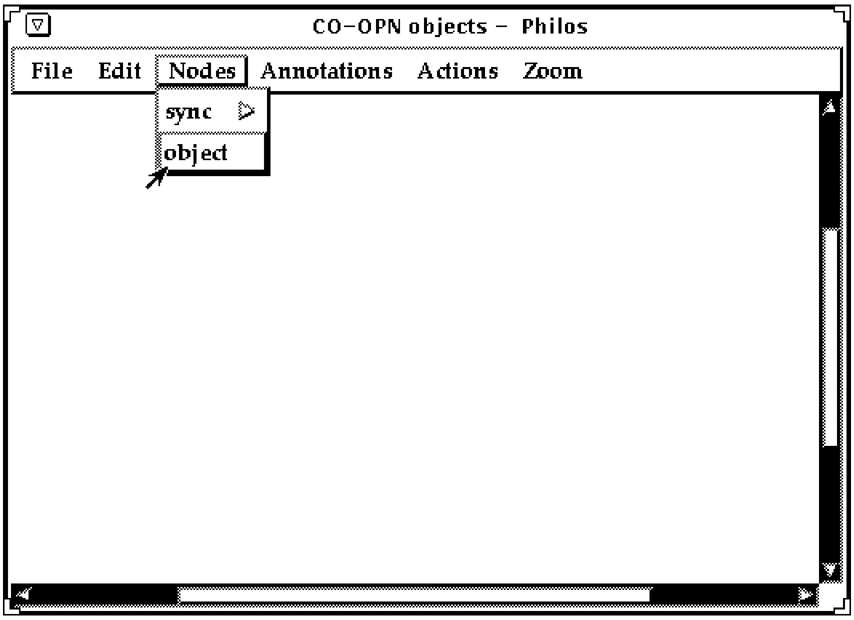


Fig. 21: Creation of a CO-OPN object

When the name of an object is given the user has to place the object with the mouse obtaining a graphical representation of the latter (see Figure 22).

4.3.2 Methods

By clicking twice on an object a menu is activated, presenting several options (see Figure 22).

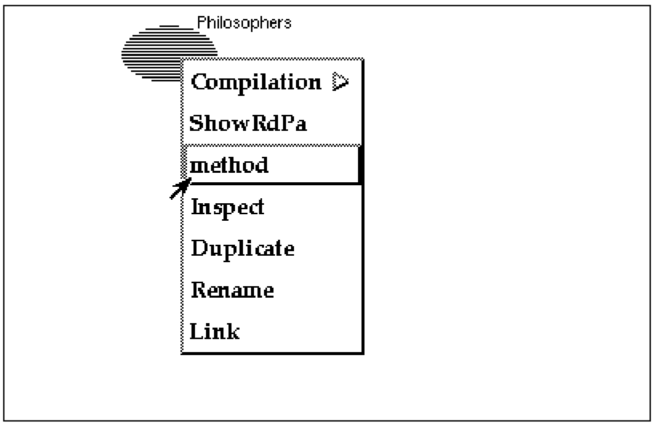


Fig. 22: Menu of a CO-OPN object

The '`ShowRdPA`' command allows to visualize the algebraic Petri net associated to the object. If the user wants to create a method, he selects the '`meth`od' command (in the same menu) and types an identifier name. The method is placed near from the CO-OPN object (at the bottom of the object), but it is also created in the window of the algebraic Petri nets associated. The CO-OPN editor will select it automatically. The description of this graphical window is presented in paragraph 4.4.

4.3.3 Signature of a method

When a method is defined, the user can define a signature. In this case, he clicks twice on it and a menu appears (see Figure 23).

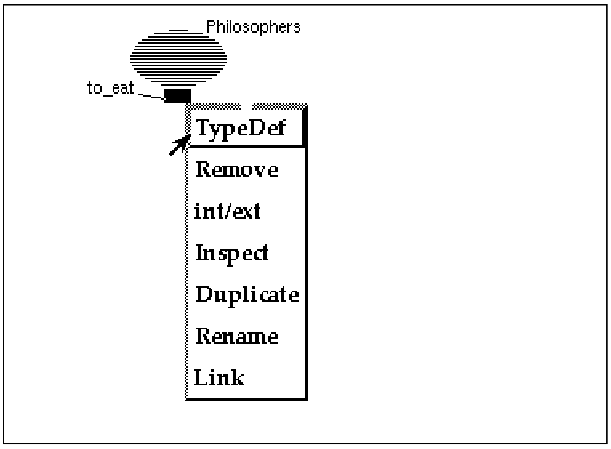


Fig. 23: Menu of a method (at the CO-OPN object level)

When the user selects the '`TypeDef`' command a window appears (see Figure 24). In this window, all the algebraic data types which are defined in the algebraic specification window are presented (cf. 4.2).

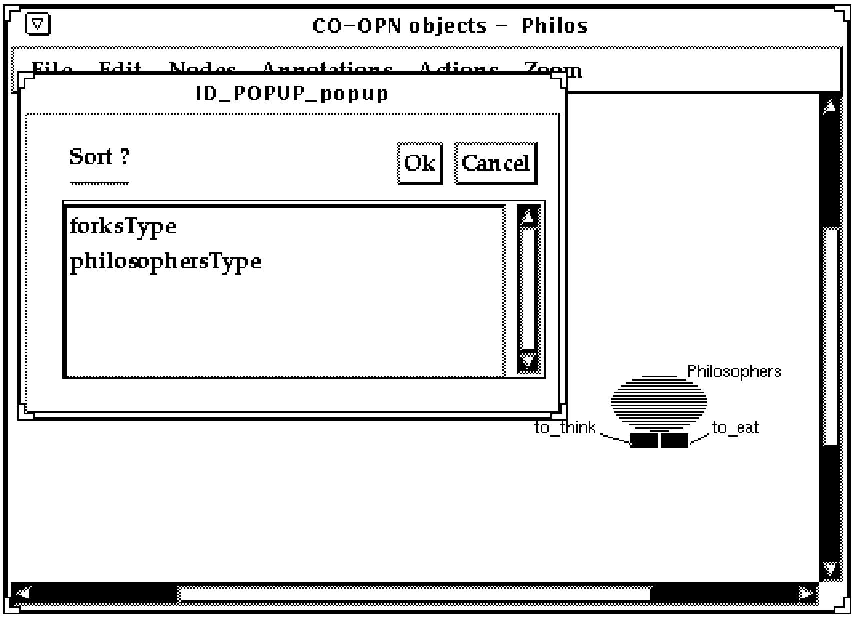


Fig. 24: Choice between all the algebraic data types

Then, he can click on the line of the chosen algebraic data type and define the signature and

the parameters (or expression) which are associated to it. When the signature is completely defined, the user will select the '`Cancel`' button.
It is not possible to correct an error in the definition of the signature. There is only the possibility to delete it completely. This can be done if the user selects another time the '`TypeDef`' menu and clicks the '`Cancel`' button, the signature of the methods and its parameters are erased.
It is possible to visualize the signature of the method in two different ways. The user can select the '`Actions`' menu in the objects window and a submenu appears (see Figure 25).

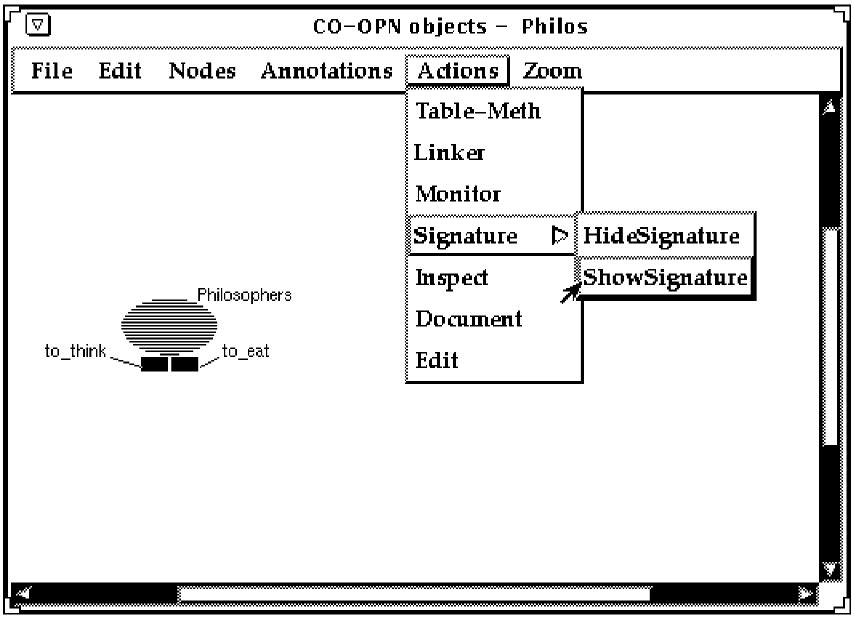


Fig. 25: Menu 'Actions' of the objects window

In the menu (see Figure 25), the user can visualize the graphical representation of the signature of the method, he just has to select the '`Signature`' command (Hide or Show). He can also use the '`Table-Meth`' command which creates an 'interactive' table. This table highlight some informations which are about the objects and the methods of the model.
An 'interactive' table allows the user to change the cells by clicking twice with the left button on the chosen cell. He could set up a new value.

4.4 Algebraic Petri net window

A window of an algebraic Petri net is a window which is associated to a CO-OPN object. It is possible to define places and transitions as it is possible with any Petri net editor (see Figure 26).

4.4.1 Transition & methods

It is important to realize the equivalence between the abstract representation of an object and its associated graph. We detailed in the previous section how a method was created. During the creation of a method from the abstract representation of an object (fig 22), this method is automatically placed on the 'network' graph which is associated to it. On the other hand, if the user creates or changes a method in the associated graph of the object, this change is also performed at the level of the abstract representation of the object.

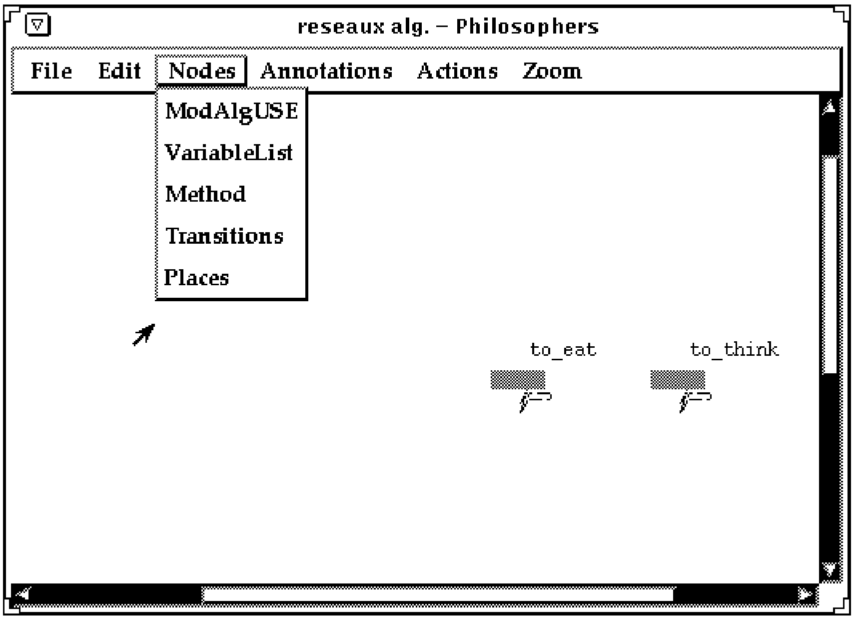


Fig. 26: 'lasses' Menu in the algebraic network window

Figure 26 shows items definable in the algebraic network. Items `'ModAlgUSE'` and `'VariablesList'` have certain particular aspects. They items variable definition algebraic module importing.

4.4.2 Places

From the selected place, the user has to click twice on the latter place to obtain a menu (see Figure 27).

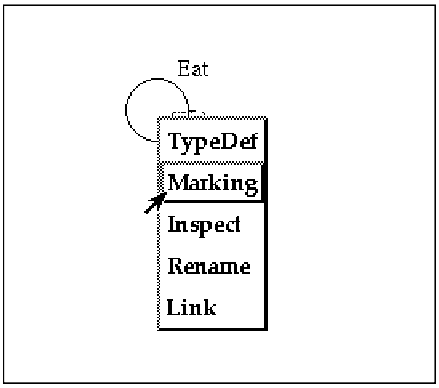


Fig. 27: Menu from a place

When the `'TypeDef'` item is selected, the user obtains a window where all the abstract data types are presented. These types were defined in the algebraic window (see Figure 24). The user can choose the type of a place by clicking twice on a type of that window. Then the user has to decide if the place is multivalued by a multiset or not. In the current version, all the places have to be multivalued. The item `'Marking'` allows the user to define the initial marking of the place.

4.4.3 Firing conditions

It is possible to duplicate methods or transitions but not the places. The transition or the method are unique in the sense that a modification of the name is performed on all their copies, but each of them can have specifying firing condition. Thus, how to change the firing condition.

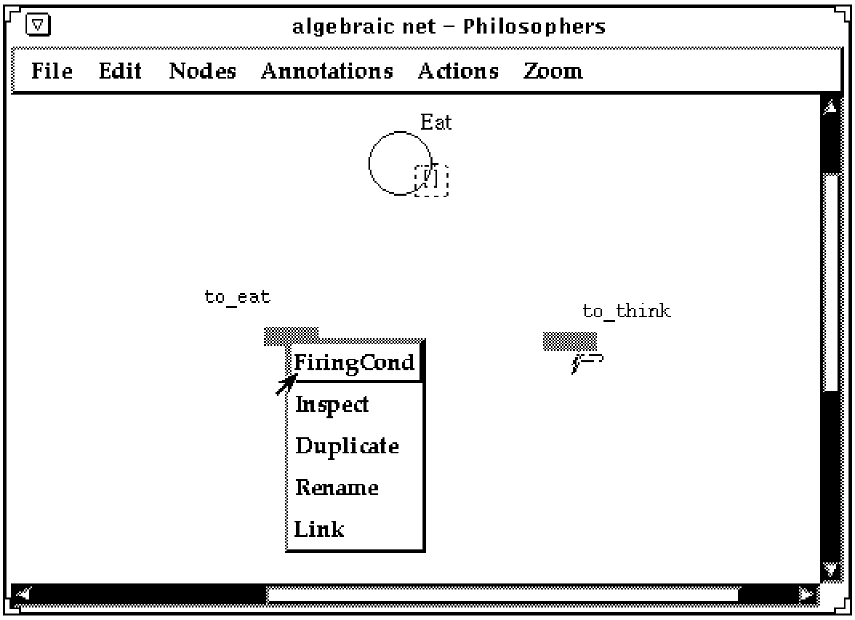


Fig. 28: Menu from a method (algebraic nets level)

If the item '`FiringCond`' is selected in the menu which is available from a method or a transition, the user can introduce the associated text to the chosen conditions.

4.4.4 Creation of an edge(arc)

How to create an arc between two objects. The user selects the item '`Link`' which is available in the 'place' menu or 'transition' menu (see Figure 27 & 28). Then, the user has to click on the destination object. He can also place some intermediate control points when he wants to choose himself the shape of the arc. .

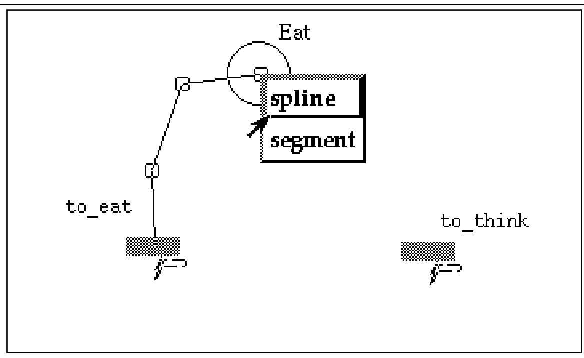


Fig. 29: Arcs (algebraic nets)

When the destination object is chosen, a menu appears which offers two options (Figure 29):

- the arc can be drawn with linear segment (option '`segment`');

- the arc can be drawn with splines through control points (option ’`spline`’).

During the creation of a method, the conditions are edited right below the method and they are initialized to empty. The user can decide to visualize , or not,, depending on the state of switch ’`Cond[ON/OFF]`’ of the ’`Actions`’ menu (Figure 30). We can note that edited conditions are the non-empty conditions except at creation.
The case of the transition is the same. The user can choose to visualize the type of places, it depends on the switch ’`[Type[ON/OFF]`’ of the ’`Actions`’ menu (Figure 30).
The ’`Table`’ command is also available in the menu of the Figure 30, it allows to obtain interactive table, which content informations on the algebraic net.

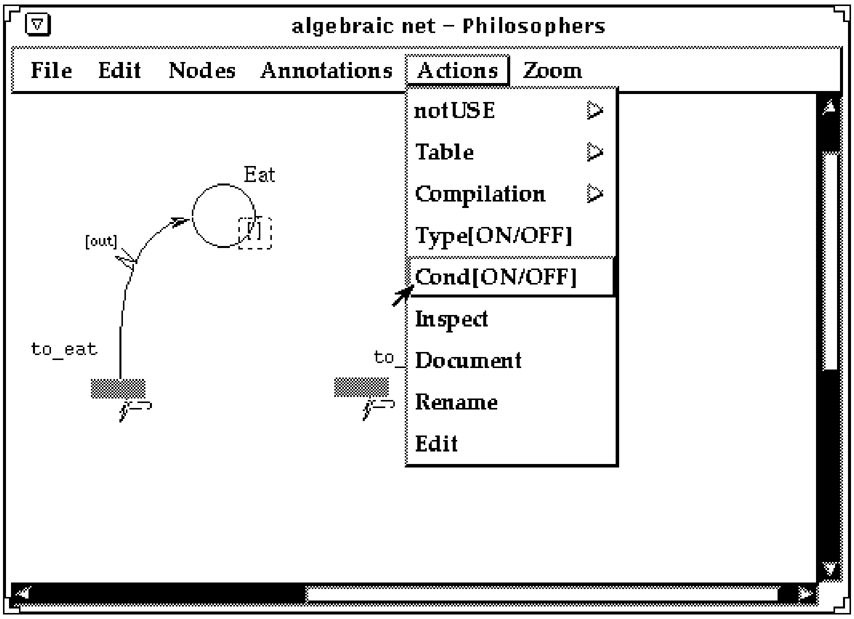


Fig. 30: ’Action’ menu in an algebraic window

We can note about the ’`Compilation`’ command in the menu of Figure 30, that the command allows to compile the Petri net but, it is divided into subcommands which have to be executed in a sequential way. If the user wants to compile with just an alone command, he will use the ’`Compilation`’ command which is available from a CO-OPN object menu (see Figure 22). The submenus allow compiling and visualizing of all the internal informations of the compilation phase.
Note:It is essential to compile the files of the algebraic specifications before to compile the CO-OPN objects.

4.5 Synchronization

In the CO-OPN language, it is possible to specify synchronisation between objects with links. These links can be created at the ’objects view’ level (cf. 4.3). We precise that these links are not arcs in the sense of Petri net arcs because they rely on two transitions (methods).
The sequence for creating an arc of synchronization is similar to the sequence for creation of an arc at the algebraic level (cf. 4.4). The user selects the ’`Link`’ command in the method menu and clicks on an other method to join them together. If a parameter is necessary, the question is asked by the interface.
In our example of Figure 30 the method ’to_eat’ is defined with the formal parameter ’p’. The user has chosen to call this method with the variable ’Name’ (calling parameter). The editor highlight this parameter on the arc of synchronization.

When the user attempts to create a second arc from a method, the editor asks for the category of synchronisation (see Figure 31). Obviously, if a synchronization is needed from a method with two or more methods, it is necessary to precise the type of synchronisation between them.

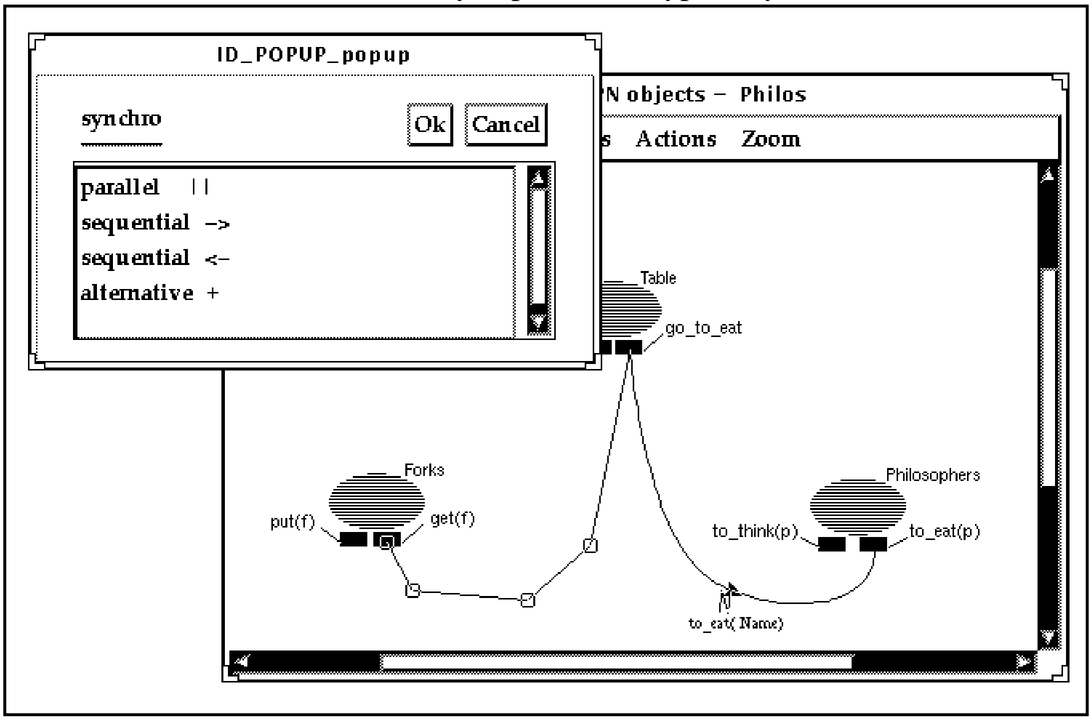


Fig. 31: Categories (type) of synchronisation

In figure 31, the type of synchronisation chosen is ’left-right sequential’, the behaviour is to first, execute the method ’to_eat’ and second, the method ’get’. In this case, the method ’go_to_eat’ of the object ’Table’ will be executed. We precise that we use a term ’f_left(Name)’ for the call of the method’get’ (Figure 32).. After the creation, the editor creates a tree with two synchronization links.

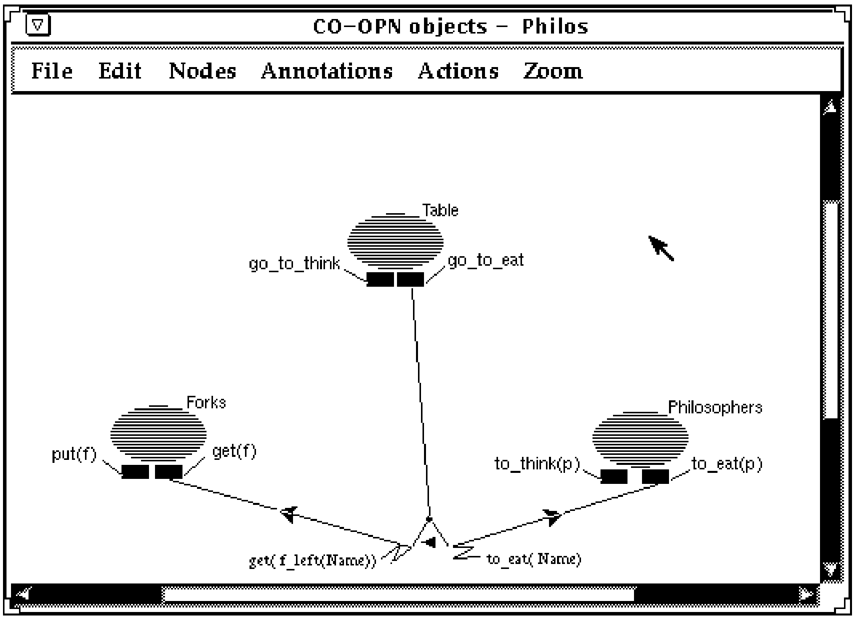


Fig. 32: View of a node of synchronisation

Unfortunately, the editor does not keep the control points defined by the user, it creates instead lines which replace synchronisation arcs. The poor result is depicted in figure 32. However the user can modify these links with the position of the synchronisation node as he wishes.
A synchronisation links can also be created by clicking twice on a synchronisation link. A menu appears (fFigure 33) in which the different type of synchronisation can be selected ('sync //, sync <-, sync ->, alternative').

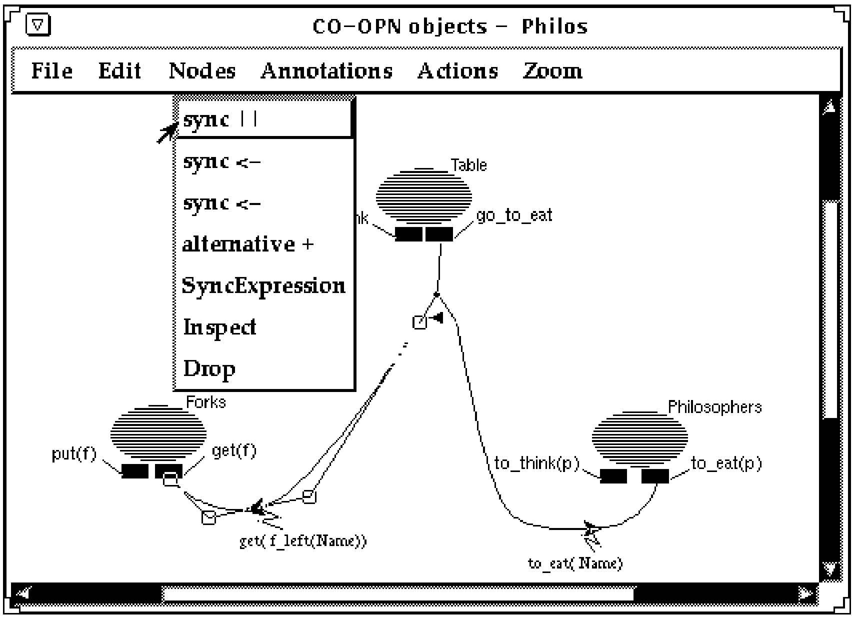


Fig. 33: Menu of a synchronisation arc

In the example, a parallel synchronisation type has been chosen (Figure 34).

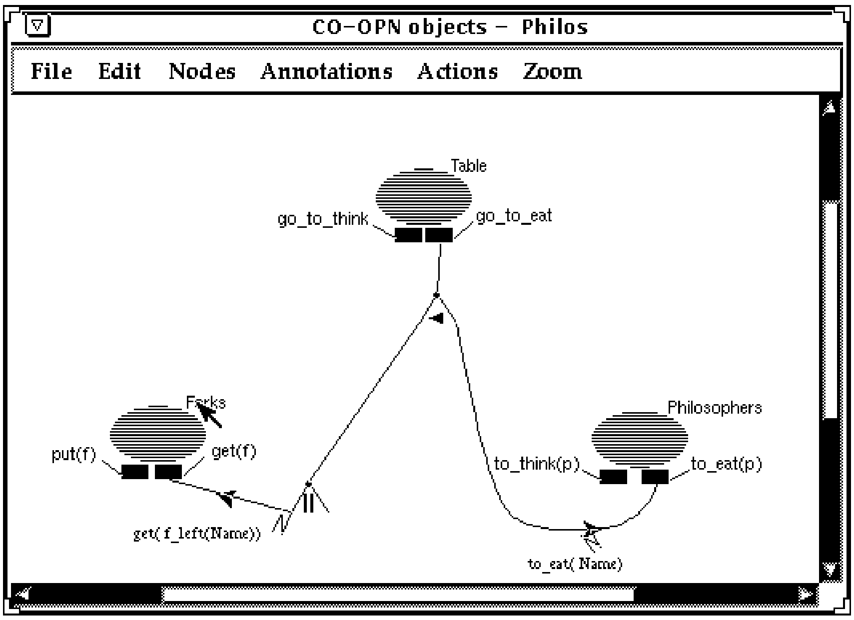


Fig. 34: Partial synchronisation tree

The figure 34 shows a node of parallel synchronisation with a single branch (partial tree). Then the user has first to click twice on this node and has to choose the command '`Link`' and then, click on the destination method. The first arc from a synchronisation node can be put at the left or right side of this node. This can be achieved by selecting the arc by moving the control point.

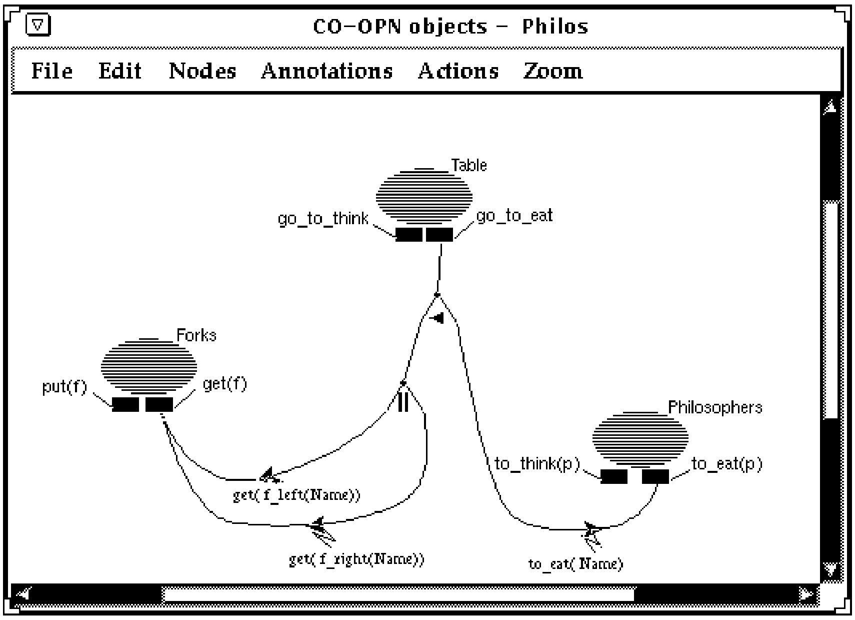


Fig. 35: A complete synchronisation tree

On the figure 35, we can view a complete synchronisation expression (tree of synchronizations). In the example, the method 'go_to_eat' can be executed if: we can execute in sequence, the method 'to_eat' of object 'Philosophers' and after, concurrently the methods 'get' with parameter 'f_left(Name)' and 'get' with parameter 'f_right(Name)'.

In other words, the sentence:

> For eat ('go_to_eat'), a philosopher must first, decide 'to eat' and
> after take concurrently his left and right fork ('get(...)')

4.6 Synchronisation from an internal transition

In the semantic of the CO-OPN language, an internal transition can have a synchronization with some methods of objects. These internal transitions must be rely to other methods with synchronisation arc.
For create a such transition, the user must first defines a method like in paragraph 4.3 or 4.4. The second action must be done in the 'objects view' window, the user will obtain the menu on the method (double click) and select the '`int/ext`' command. A graphical result is to change the full black box of method to an white box (fig 36), and the method is now consider like an internal transition. In the algebraic window, the graphical form of this method is modified too.
We must precise that for change a method in internal transition, two conditions must be required: the signature of this method must be empty and it can not be a destination method for

a synchronisation arc.

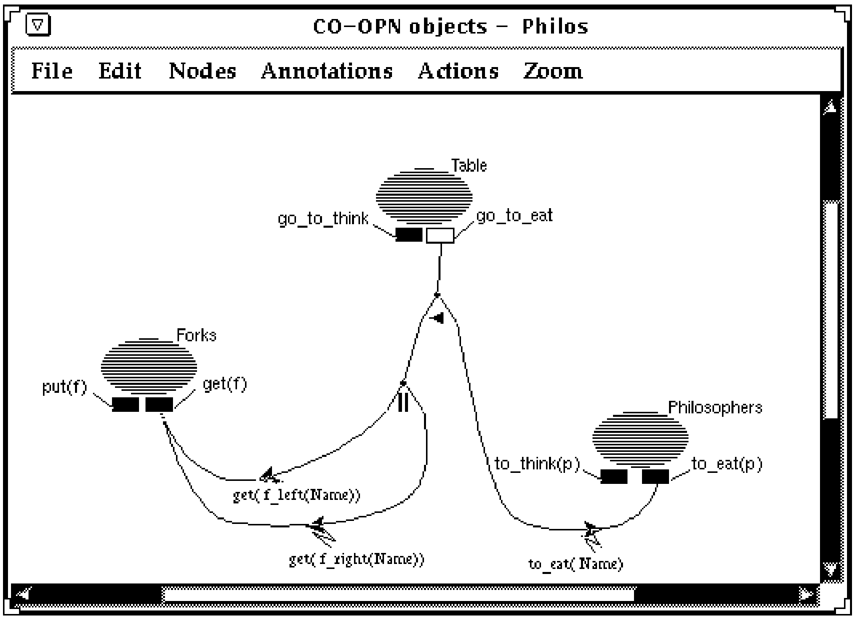


Fig. 36: Synchronisation from an internal transition

The figure 36 shows the transformation of the method 'manger' in an internal transition. This case is incorrect in the complete system of 'Philosopher table', it is used only to illustrate this paragraph.

4.7 Full Description of Philosophers Table

4.7.1 Introduction

The following paragraphs shows the complete system of 'Table of philosopher'. The paragraphs 4.7.2 and 4.7.3 are the textual form of the specification that is necessary to import in the editor to develop the application. The 4.7.4,4.7.5 and 4.7.6 paragraphs show the graphical representation of each object in the system. At the end, the paragraph 4.7.7 shows the 'objects view' with all synchronisation arcs between objects.

4.7.2 PhilosophersType specification

```
spec PhilosophersType;
sorts philosophersType;

generated by

Socrate  : -> philosophersType;
Platon   : -> philosophersType;
Aristote : -> philosophersType;

end
```

4.7.3 ForksType specification

```
spec ForksType;
```

```
use PhilosophersType;

sorts forksType;

generated by

f0 : -> forksType;
f1 : -> forksType;
f2 : -> forksType;

operations

f_left  : philosophersType -> forksType;
f_right : philosophersType -> forksType;

axioms

left1 : => f_left(Socrate ) = f0;
left2 : => f_left(Platon  ) = f1;
left3 : => f_left(Aristote) = f2;

right1 : => f_right(Socrate ) = f1;
right2 : => f_right(Platon  ) = f2;
right3 : => f_right(Aristote) = f0;

end
```

4.7.4 Forks Object

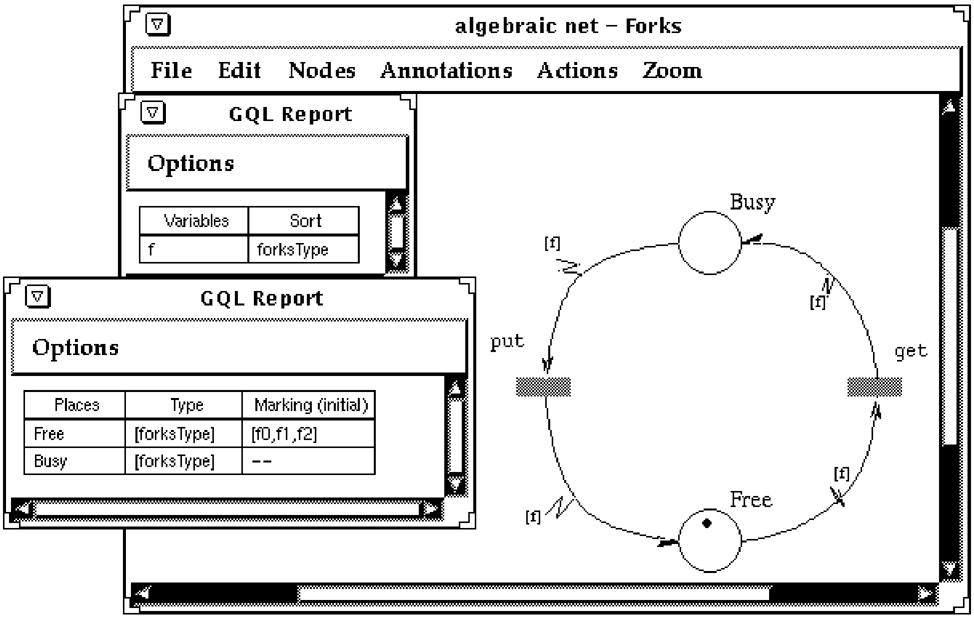


Fig. 37: Algebraic net of the 'Forks' Object

4.7.5 Philosophers Object

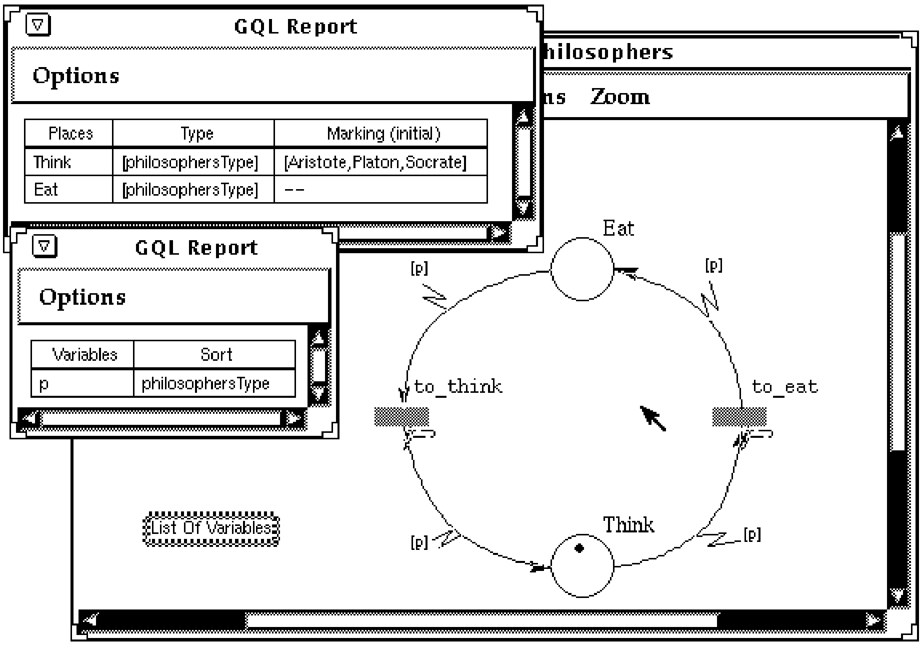


Fig. 38: Algebraic net of the 'Philosophers' Object

4.7.6 Table Object

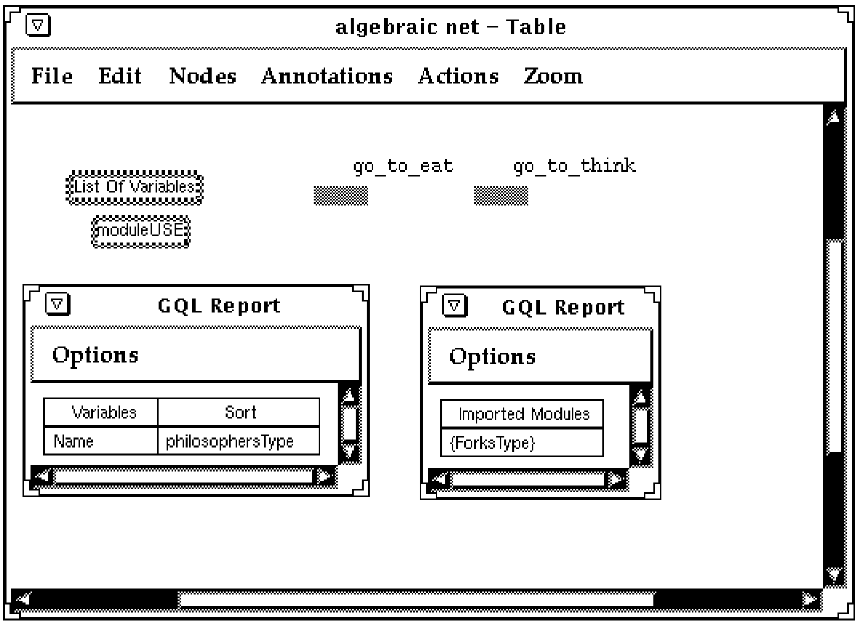


Fig. 39: Algebraic net of the 'Table' Object

4.7.7 The abstract view of the system

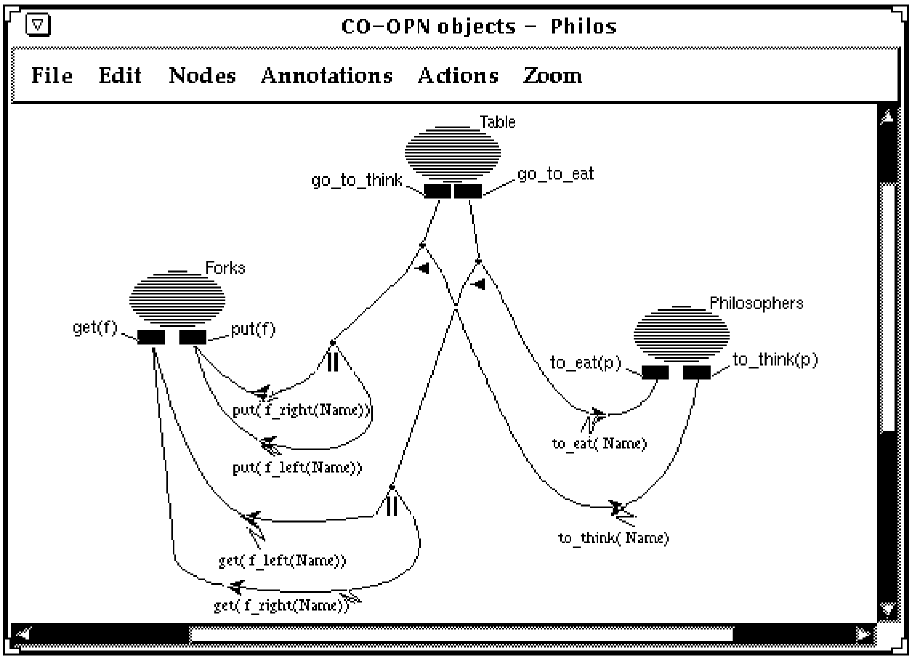


Fig. 40: The complete representation of the 'Philosopher Tables' system

Chapter 5

Compiler

5.1 Introduction

The compiler and the simulator compose the main two tools of the environment. The compiler uses textual representations of the system modelled and generates Prolog code. The textual description of the object specification can be produced either through the graphical editor or it can be hand-written by the user in the CO-OPN specification language. The code produced is interpreted by the execution monitor, or *engine*, in conjunction with the simulation environment to simulate the system modelled. The overall environment is depicted in Figure 41:

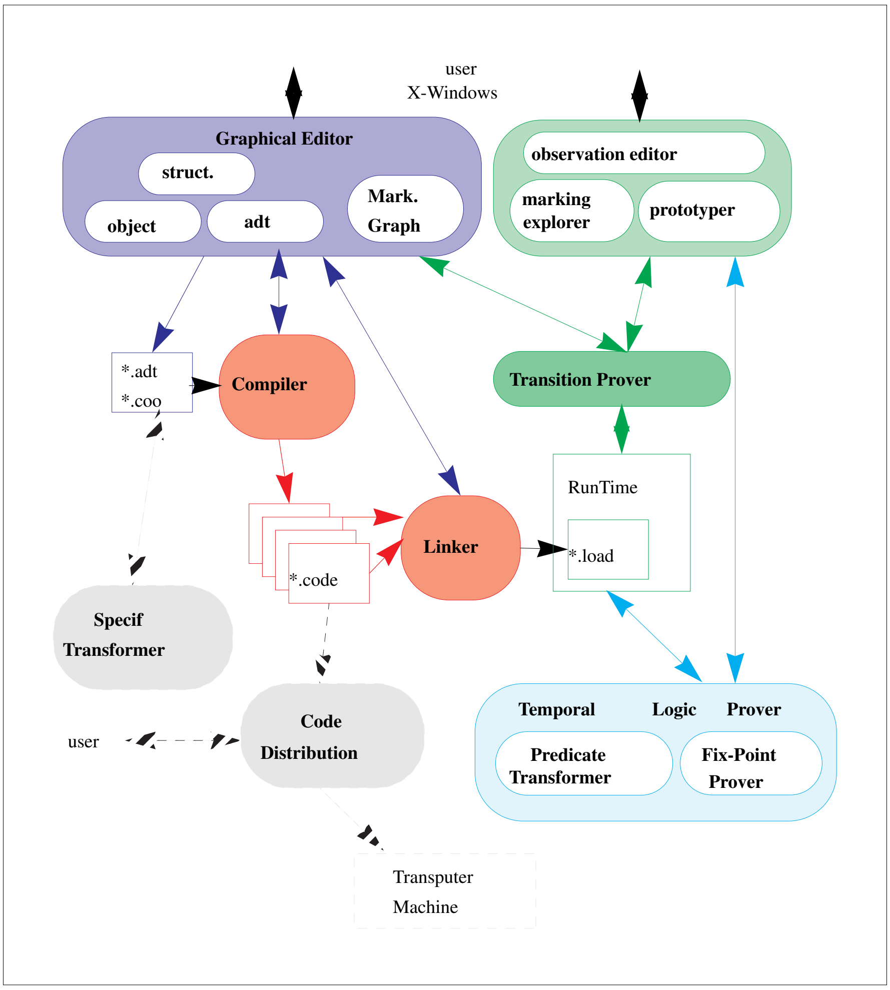


Fig. 41: Overall structure of SANDS environment

CO-OPN specifications are translated in Prolog code and clause executions are handled by the execution monitor. These translations differ whether they are connected to an algebraic data specification or to an object specification. The translation process is performed only on non-generic specifications and instantiated or ground specifications are free of identification name collision (see 5.4. modularity). An item is specified by its name and eventually its arity but its sort is not significant. The overloading characteristics are nevertheless supported.

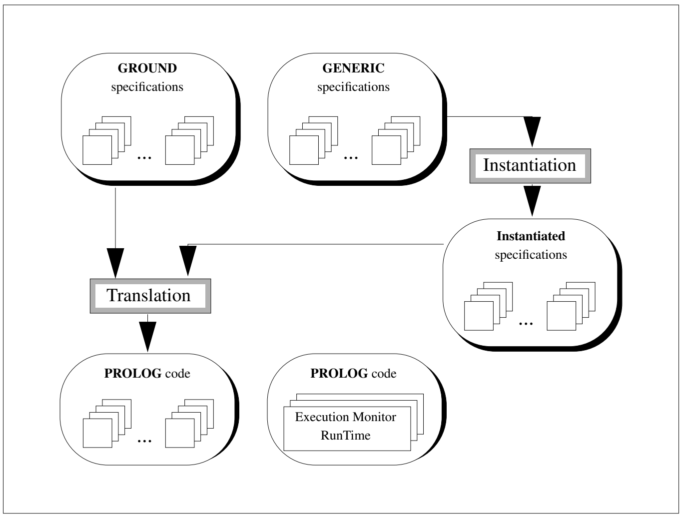


Fig. 42: Instantiation and translation process

5.2 Supporting example

In order to exemplify some particular points we will use throughout this section the modelling of the Alternating Bit Protocol (ABP).

5.2.1 The protocol

The alternating bit protocol (ABP) allows two communicating entities to alternatively exchange messages over an unreliable transmission medium. Messages can be lost or altered. The existence of an altered message detection mechanism is assumed and a classical time-out retransmission mechanism is used to cope with the problem of lost messages. When an entity receives a corrupted message it rejects it, reducing at the same time the problem of corrupted messages to that of lost messages.

Although the original ABP was designed to achieve reliable full-duplex data transfers, we assume the existence of only one sender who sends messages to the only receiver who acknowledges the correct receipt of the sender's messages. We will also assume that the transmission lines consists of two unreliable FIFO queues.

Each time the sender initiates the transmission of a message, it starts a timer. If no acknowledgement of the delivery is received within the given delay of time by the sender, the message is retransmitted. It is to be noted that the time-out mechanism can generate duplicated messages at each side of the transmission medium if the time-out delay has been set too short.

In order to solve the problem of duplication, messages are sequentially numbered with modulo-2 sequences. Consequently, each message, apart from containing error detection information, contains also a flag bit that serves as an identification number. If the receiver gets a message with a flag bit equal to the flag bit of the last acknowledgment it has sent, the receiver then detects a duplicated message. In order to allow the sender to synchronize, the receiver also returns acknowledgment for duplicated messages. A duplicated acknowledgment is received by the sender if its flag bit is different from the flag bit of the acknowledgment that the receiver is ready to accept.

5.2.2 The modelling

The ABP is structured with the objects *Transmitter*, *EtherIn*, *Receiver* and *EtherOut* (see Figure 43). The *EtherIn* and *EtherOut* objects model respectively the channel where the *Transmitter* sends messages and the *Receiver* sends acknowledgments.

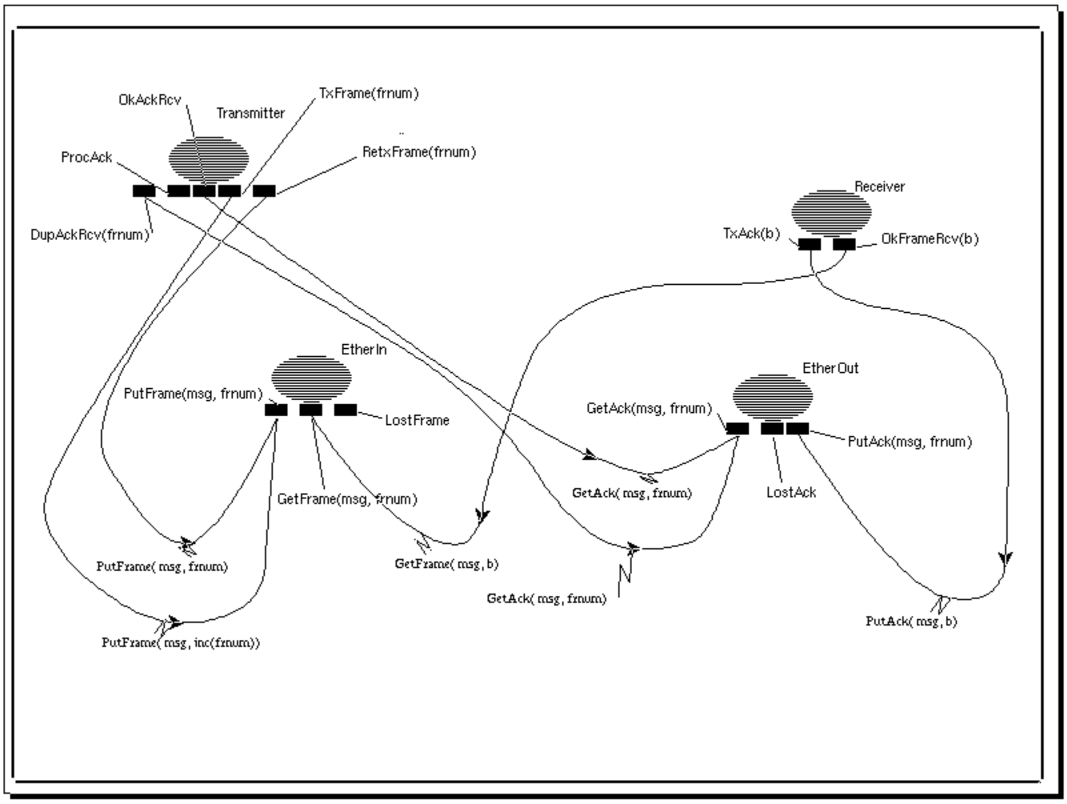


Fig. 43: The Alternating Bit Protocol model

5.3 Algebraic module

The similarity between algebraic specifications and Horns clauses induce a straightforward translation of these specifications in Prolog. Due to the evaluation order importance of the terms, the specification equations are viewed as rewrite rules oriented from left to right. These rules are translated in Horns clauses.

Translation of equational axioms in Horn clauses

The axioms accepted by the language are the conditional positive axioms. An equality of the form $f(X_1,...,X_n) = Y$ is associated to the predicate $f(X_1,...,X_n,Y)$. These transformations are valid because the unification process preserves the equality properties of reflexivity, commutativity and associativity.

```
Ex:

Let us consider the following axiom

                          c1(x)=c2(x) => f(x)=g(x)

It can be rewritten with variables r1,r2 as

               (c1(x)=r1) ∧ (c2(x)=r2)∧(r1=r2) => f(x)=g(x)

Which gives in Horn clause

               c1(x,r1)∧c2(x,r2) ∧ r1=r2 ∧g(x,r3) => f(x,r3)

It is equivalent to

                 c1(x,r1) ∧ c2(x,r1) ∧ g(x,r3) => f(x,r3)
```

Algebraic symbols

The table below gives the relation between an algebraic symbol and its translation in Prolog:

Sort of the term	**Prolog correspondence**	**Syntax**	**Example**
Variable	Variable	_number	_1251
Operator	Predicate	OperatorName	add
Generator	Functor	GeneratorName	succ

Modularity

A term belongs to the module where it is defined. This membership is specified in the term's name by the construction *nameofthemodule_nameoftheterm*.

```
Ex: Transmitter_TxFrame, Bool_true, AltBit_inc
```

Algebraic terms

Algebraic terms are flattened using the *depth_first* strategy. This process allows the interpretation of algebraic axioms.

Let *f,g, h* be functions, *X, Y* variables and *a* a generator. The term

```
                              f(g(a(Y),X),h(Y))
is translated as

               g(a(_Y),_X,_r1) ∧ h(_Y,_r2) ∧ f(_r1,_r2,_r3)
```

Algebraic axioms

The combination of the preceding techniques are used to translate algebraic axioms. The predicate `nil` is defined as the fact `nil`.

```
Ex:
           n1: IsEmpty(ff) = false => numframe(add(ff,fr)) = numframe(ff);

           is translated in

           FiFo_get(FiFo_add( _1016 , _1028 ), _1284 ) :-
                       FiFo_IsEmpty( _1016 ,Bool_false) ,
                       FiFo_get( _1016 , _1284 ) , nil.

Ex: The complete Prolog code of the module FiFo

           spec FiFo;
           use Frame, Bool, AltBit, Message;
           sorts fifo;
           generated by
           empty : -> fifo;
           add : fifo, frame -> fifo;
           operations
           IsEmpty : fifo -> bool;
           rm : fifo -> fifo;
           get : fifo -> frame;
           numframe : fifo -> altbit;
           axioms
           e1 : => IsEmpty(empty) = true;
           e2 : => IsEmpty(add(ff,fr)) = false;
           r1 : => rm(empty) = empty;
           r2 : => rm(add(empty,fr)) = empty;
           r3 : IsEmpty(ff) = false => rm(add(ff,fr)) = add(rm(ff),fr);
           g1: => get(add(empty,fr)) = fr;
           g2: IsEmpty(ff) = false => get(add(ff,fr)) = get(ff);
           n1: IsEmpty(ff) = false => numframe(add(ff,fr)) = numframe(ff);
           n2: => numframe(add(empty,comp(msg,num))) = num;
           where
           ff : fifo;
           fr : frame;
           msg : message;
           num : altbit;
           end

           is translated in

           FiFo_IsEmpty(FiFo_empty,Bool_true) :- nil.
           FiFo_IsEmpty(FiFo_add( _991 , _1003 ),Bool_false) :- nil.
           FiFo_rm(FiFo_empty,FiFo_empty) :- nil.
```

```
FiFo_rm(FiFo_add(FiFo_empty, _998 ),FiFo_empty) :- nil.
FiFo_rm(FiFo_add( _1072 , _1084 ),FiFo_add( _1389 , _1084 )) :-
            FiFo_IsEmpty( _1072 ,Bool_false) ,
            FiFo_rm( _1072 , _1389 ) , nil.
FiFo_get(FiFo_add(FiFo_empty, _840 ), _840 ) :- nil.
FiFo_get(FiFo_add( _1016 , _1028 ), _1284 ) :-
            FiFo_IsEmpty( _1016 ,Bool_false) ,
            FiFo_get( _1016 , _1284 ) , nil.
FiFo_numframe(FiFo_add( _1016 , _1028 ), _1284 ) :-
            FiFo_IsEmpty( _1016 ,Bool_false) ,
            FiFo_numframe( _1016 , _1284 ) , nil.
FiFo_numframe(FiFo_add(FiFo_empty,Frame_comp( _1072 , _843 )),
_843 ) :- nil.
```

5.4 Object modules

The code generated from the translation cannot be used directly to perform a simulation. The semantic rules of the CO-OPN formalism, describing the evolution of a system specification, are needed. These rules have been transcribed in a so-called RunTime.

Net structure of an object

The informations about the places, methods or internal transitions of an object are generated as follows:

```
Ex:The object Receiver

            places(Receiver,[FrameRcv]).
            methods(Receiver,[Receiver_TxAck,Receiver_OkFrameRcv]).
            internals(Receiver,nil).
```

Axioms of an object

The predicate associated to a CO-OPN object axiom (axiom/7) has the following structure:

- The name of the module where the axiom is defined;
- The name of the axiom;
- The code used to call the method or the internal transition;
- The synchronization expression;
- The condition (built as a conjunction of equalities);
- The input marking;
- The output marking.

Method Call

The call of an object method is composed of a couple `(code,methodname(`$t_1,...,t_n$`))` where code is a list of terms ended by the `nil` predicate. This list flattens the parameter's term expression before evaluation. The result of each of these evaluations are the terms $t_1,...,t_n$.

```
Ex:
            The term TxAck(V) where V is a variable is translated as:
            (nil , Receiver_TxAck( _1574 ))

            TxFrame(zero) as:
            (nil,Transmitter_TxFrame(AltBit_zero))

            TxFrame(inc(zero)) as:
            ((AltBit_inc(zero),_15),nil),Transmitter_TxFrame(_15))
```

Marking

The marking of an object is represented by a list, the elements of which are the place's contents. The content of a place is expressed by a multi-set construction by means of the functors cempty/0, cid/1 and cplus together with a code part. As usual the code part flattens the terms, if needed, to enable their evaluation.

Ex:

If terms t_1, t_2 and t_3 needn't code parts, the multi-set $[t_1,t_2,t_3]$ is translated as:
(nil, cplus(cid(t_1),cplus(cid(t_2),cid(t_3))))

```
            Let us consider FrameRcv([add(ffrcv,comp(msg,inc(num)))]) where
            ffrcv, msg and num are variables. The multi-set of the place is
            translated as:
            ((AltBit_inc(_0,_1),nil),
                        cplus(cid(FiFo_add(_2,Frame_comp(_3,_1))),cempty))

            The initial marking of the object Receiver:
            initial(Receiver,[(nil , cplus(cid(FiFo_empty),cempty))])
```

```
Complete ex: The Receiver object

            places(Receiver,[FrameRcv]).
            methods(Receiver,[Receiver_TxAck,Receiver_OkFrameRcv]).
            initial(Receiver,[(nil , cplus(cid(FiFo_empty),cempty))]).
            internals(Receiver,nil).

            axiom(      Receiver,
                        a396,
                        (nil , Receiver_TxAck( _1620 )),
                        cid((nil , EtherOut_PutAck( _1687 , _1620 ))),
                        nil,
                        [(nil , cplus(cid(FiFo_add( _1894 ,
                              Frame_comp( _1687 , _1620 ))),cempty))],
                        [(nil , cplus(cid( _1894 ),cempty))]).

            axiom(      Receiver,
                        a394,
                        (nil , Receiver_OkFrameRcv( _1620 )),
                        cid((nil , EtherIn_GetFrame( _1687 , _1620 ))),
                        nil,
                        [(nil , cplus(cid( _1749 ),cempty))],
                        [(nil , cplus(cid(FiFo_add( _1749 ,
```

```
Frame_comp( _1687 , _1620 ))),cempty))]).
```

Method parameter finding

The unification process of Prolog and the backtrack mechanism is used by the execution monitor to find the set of terms within place contents of the current marking that can be unified with the input terms of a given method. Each of these terms which do not satisfy the condition attached to the method are rejected.

Ex:

Let us consider the net of Figure 44. The input marking of the method is *succ(X)* and its condition *prime(X)*. In order to be firable, the following requirement has to be fulfilled: ∃ Y ∊ P such that Y = succ(X) and prime(X) = true. The content of the place P is supposed to be [zero, succ(succ(zero)), succ(succ(-succ((zero))), succ(succ(succ(succ(succ(zero)))))].

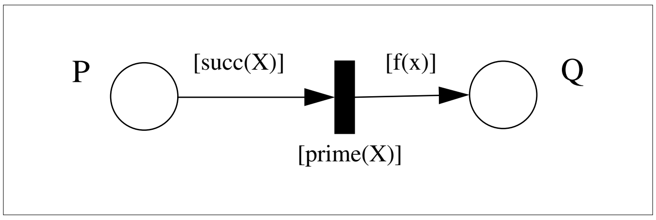


Fig. 44: A simple net

The following set of terms enables the occurence of the method:

```
{           Y = succ(succ(succ(zero))),
                        X = succ(succ(zero)),
            Y = succ(succ(succ(succ(zero))),
                        X = succ(succ(succ(zero)))  }
```

Chapter 6

Simulator and execution monitor

6.1 Simulator

6.1.1 Introduction

The simulator allows users to perform model specifications by using the classical features of a window manager (windows, buttons, scrolling list, etc.). However, the graphical features such as the move of 'tokens' is not available. Instead, symbolic representations of tokens using algebraic terms are used (possibly with free variables). These representations are computed with resolution techniques. Apart from the simple or structured algebraic nets, it is also possible to perform the construction and simulation of classical Petri nets. This is achieved by specifying in a first step the notion of token (see Figure 45). The compiler is then informed (by an option) that the specification is related to simple nets and automatically operates the necessary transformations. In the same way the underlying Petri net of a structured algebraic specification (its skeleton) can easily be obtained and simulated.

spec BlackToken;
sorts blacktoken;
generated by

tok : -> blacktoken;

Fig. 45: The *BlackToken* specification

The simulation tool has the notion of visibility/availability. The user can specify the set of objects, places and transitions he wants to concentrate on. When an item is hidden all information about it is masked. This feature allows the user to focus attention on a particular part of the model.

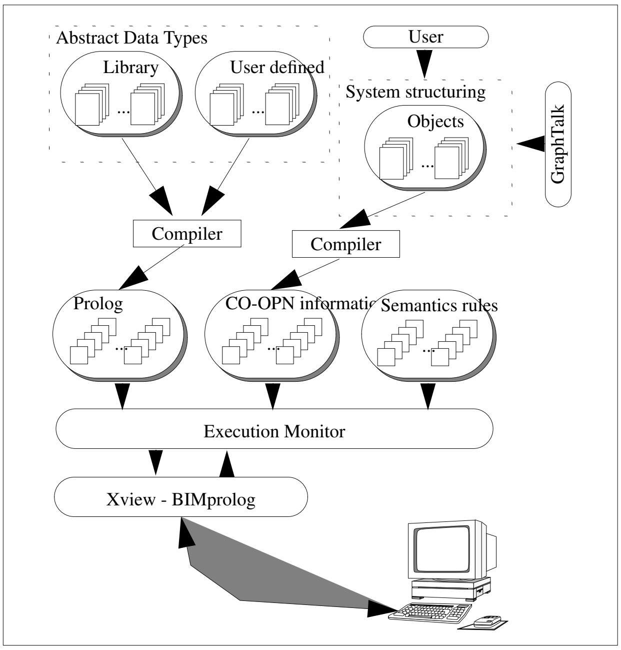


Fig. 46: The environment of the simulator

Two main ways of interaction are offered to the user. The first one is a step by step simulation. The engine computes the set of firable transitions together with their parameter instantiations (as loose as possible). The user is then asked to choose a transition in this set. Once a transition is selected the tool shows the possible parameter instantiations. The user can inspect the transition, or choose another, and the engine performs the computation of a new step in order to propose a new set of transitions/parameters from the output marking of the last selected transition. At any moment of the simulation, the user can choose another current marking, inspect a particular marking, ask for a transition information such as its input and output marking and its parameters.

In the second mode of interaction the user asks questions about the non-sequential behaviour of the system in the form of a synchronization constraint among events (methods or transitions). In the same way the engine calculates the set of events/parameters which checks the given constraint. Bounds on the sequential, parallel or non-deterministic constraint can be fixed. Set of markings can be uniquely represented using free variables on specific domains, sub-set of markings are automatically computed when testing the availability of transitions from these sets of markings. At any moment the history of a simulation can be saved or loaded.

6.1.2 Main window

This window (see Figure 47) named *CO-OPN Monitor* contains two different parts.

The lower part of the window is used to set the visibility of the items of the specification. This part includes the three scrolling lists **Objects**, **Places** and **Events** and allows to choose what will be observed or hidden relatively to the set of objects, places and events. The list of items displayed in the scrolling lists **Places** and **Events** are relative to the currently selected object. The currently selected object is surrounded by a black rectangle and its name is specified in the field **Setting of object:**. An item of a scrolling list preceded by a black dot means that it is observed. By clicking on an item of a scrolling list its visibility is changed. In Figure 47 the informations about object *EtherOut* won't be displayed and the informations about places *AckRcv* and the event *TxFrame* of the object *Transmitter* will also be hidden. The buttons of the form (Drop all) and (Observe all) (below the scrolling lists) should be self explanatory.

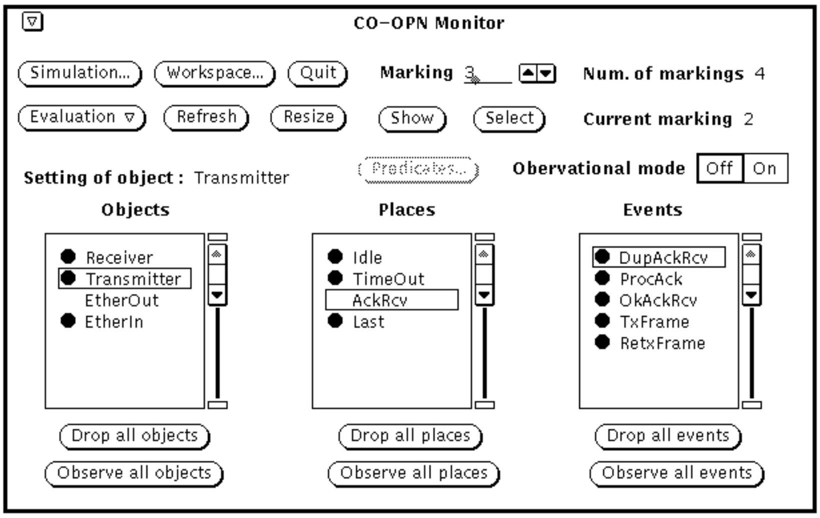


Fig. 47: Main window: CO-OPN monitor

The upper part of the window is partially involved in the markings management and is the root access to the other windows. In this part of the main window the button (Show) displays the marking whose number is specified in the numeric field **Marking** without changing the current marking while button (Select) does the same but also changes the current marking with the selected marking of the numeric field **Marking**. The object markings are displayed in devoted windows (see 6.1.3). The button (Refresh) redisplays the current marking and button (Resize) - hides the lower part of the main window.

Button (Quit) ends the simulation session.

- **Observational mode**

  The settings made in this choice button influence the way the step information is calculated. This step includes the input marking, the method and its parameters and the output marking. Roughly said, if **Observational mode** is set to **On** all redundant steps are not inserted where the term *redundant* must be clearly specified.

  A step may be viewed as generic and it can generate a set of steps if free variables occur in it. A step is redundant in the following circumstances:

    During the process of calculation of steps, a step can be the same as an already existing one. This can occur if for example the user asks the same question twice (see 6.1.4 and 6.1.5);

    The newly calculated step can be an instantiation of one already found.

  However, it is worth noting that even if two steps are identical in an observational point of view, the way these steps operate can be entirely different.

6.1.3 Marking windows

These windows are used to display the marking of objects. Each object is associated to a particular window whose label is the name of the object followed by the marking number currently displayed. In Figure 48 each object marking is relative to the marking number 4. The *EtherOut* window is not displayed since the object was hidden and the content of place *Ack-Rcv* of *Transmitter* is not displayed for the same reason.

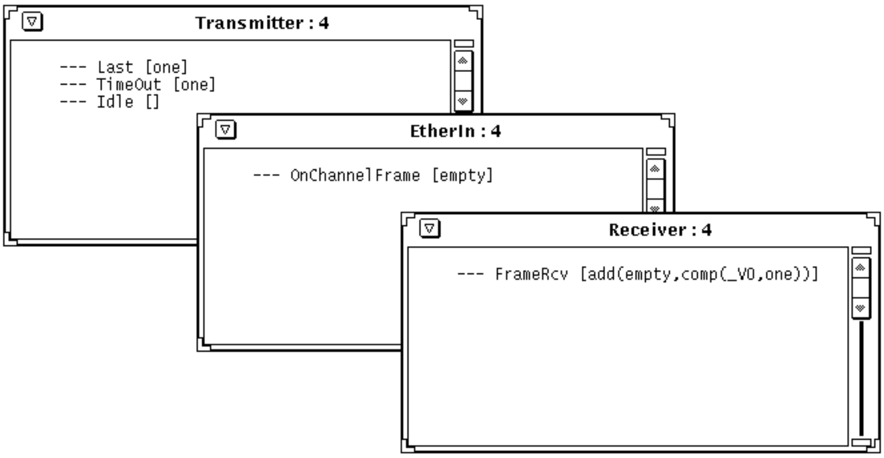


Fig. 48: Marking windows

Due to the use of variables within terms of place's contents, a marking can be viewed as a generic marking: each instantiation of the free variables generates a particular marking (see Figure 49).

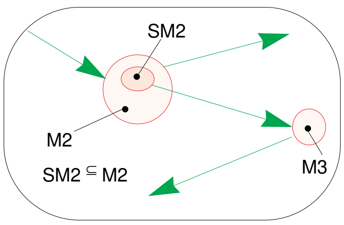


Fig. 49: Generic marking

6.1.4 CO-OPN Evaluation window

This window allows control over the evolution of the system. The expression specified in the **Synchronization expression** field of Figure 50 is a constraint expression involving methods of objects with the notions of sequential, parallel and alternative (ref. au language). This expression is used to compute the set of markings which can be reached from the current marking with the specified synchronization. The parameters displayed in the fields **Depth's bound**, **Parallelism's bound** and **Width's bound** set a limit to respectively the number of steps from the current marking, the number of transitions which can be fired simultaneously in a single step and the number of successors of a marking (see Figure 50). The setting to zero of a parameter means no limitation. The selection of the button (Start) activates the computation. When no constraint is required (as Figure 50) the synchronization expression must be set to a 'word' whose first character is '_'.

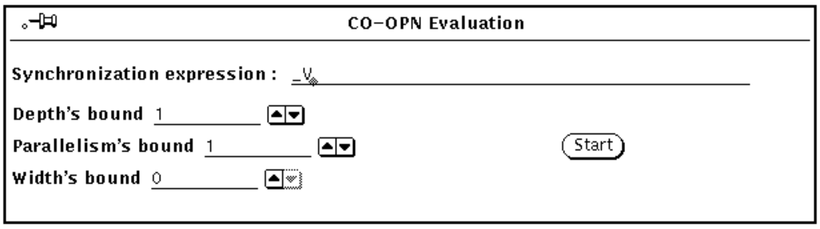
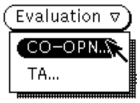


Fig. 50: Window associated to 

of *CO-OPN Monitor*

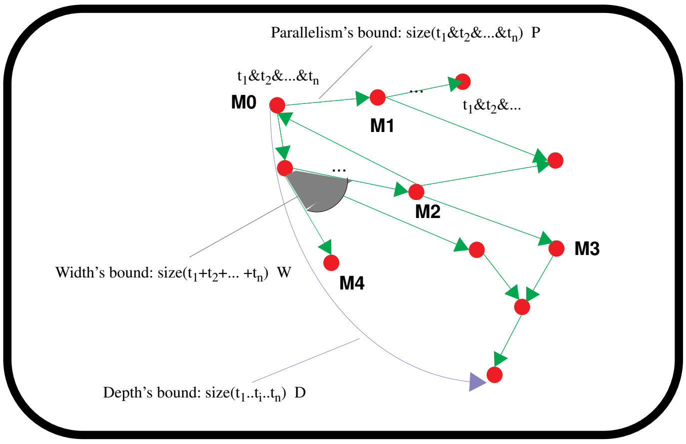


Fig. 51: Bounds associated to an evaluation

```
Ex: Synchronization expression
```

`sim(Transmitter_TxFrame,EtherIn_LostFrame)` can occur from the current marking if it's possible to fire simultaneously the methods *TxFrame* and *LostFrame*.

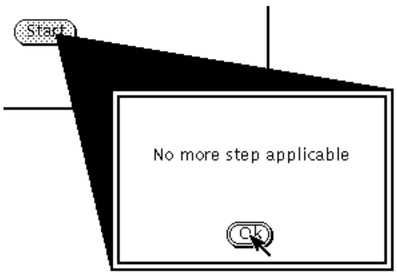


Fig. 52: Message at the end of an evaluation

6.1.5 Simulation window

The window of Figure 53 proposes a simplified way to make the system evolve. In this mode, a step by step simulation is performed where each step is composed of a single method. As a result, only the setting of the parameter **Width's bound** stays meaningful (see Figure 50).

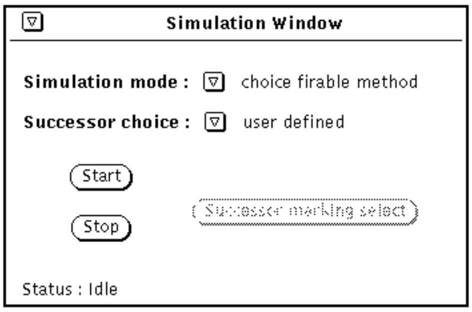


Fig. 53: Window associated to (Simulation...) of *CO-OPN Monitor*

The **Simulation mode** field (see Figure 54) is relative to the way the successors of the current marking are calculated.

When **Simulation mode** is set to **choice firable method**, the user has to choose from a set of methods, together with their parameters, which are known to be firable from the current marking. Therefore, in this mode, the engine calculates a set of firable methods and the user chooses within this set of possibilities. The choice is made with the help of the *method* window and the *transition monitor* window (see 6.1.6 and 6.1.7).

When **Simulation mode** is set to **test firable method**, the user initially chooses a method (see 6.1.6) and then the engine calculates whether the method is firable in which case the possible set of parameters are also calculated. This mode is useful when a method, or a set of methods, is/are known to have an infinite number of successors and that a width's bound limitation is nevertheless undesired.

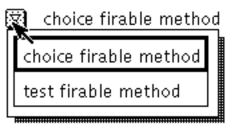


Fig. 54: *Simulation mode* field

The **Successor choice** field (see Figure 55) is related to the choice of a successor of the current marking.

When this field is set to **user defined**, the user selects or inspects a successor by means of the scrolling list **List of transitions** of the *Transitions monitor* window (see 6.1.7) where the methods together with their parameters are displayed. The confirmation of a selection is done by the (Successor marking select) button.

When the **Successor choice** field is set to **randomly**, the successor marking is chosen by the environment.

Fig. 55: *Successor choice* field

The simulation is activated by the Start button. The settings and also the current markings can be changed during a simulation. The Stop button closes the *Simulation window* and stops simulation. The simulation status is displayed at the left hand corner of the window.

The set of methods considered by the engine are those which have not been hidden by the user (see 6.1.2). Therefore a hidden method is never proposed even if it could have been firable.

6.1.6 Method Windows

A window is associated to each object and object methods are displayed as buttons. This set of windows, called *method windows*, are displayed when a step by step simulation is performed (see Figure 56). The settings relative to the visibility of objects and their methods are applied to the method windows. Therefore, if an object is hidden its associated window will not be displayed. Similarly if a method of a non-hidden object is masked, the method is not available to the user. The windows of Figure 56 show a simulation where the only visible objects are *Transmitter* and *Receiver*. If **Simulation mode** is set to **choice firable method** (see Figure 53 and Figure 54), a greyed method means either that this methods is not firable from the current marking or that the method is hidden. If **Simulation mode** is set to **test firable method**, a greyed method means that the method has been hidden.

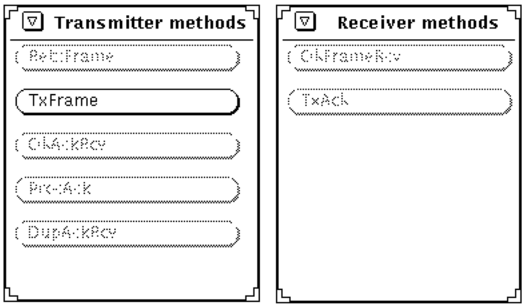


Fig. 56: Simulation mode set with *choice firable method*
Set of firable methods proposed

Figure 56 and Figure 57 show a step of a simulation where **Simulation mode** is set to **choice firable method** and **Successor choice** is set to **randomly**. Figure 56 shows that a frame can be transmitted (Clear TxFrame button). The other methods are not firable. When the button TxFrame is selected the result is shown in Figure 57 where the frame can be retransmitted (RetxFrame) by the time-out mechanism or received by the Receiver (OkFrameRcv). Due to the masking of the *EtherIn* object, the lost frame in the channel cannot be simulated.

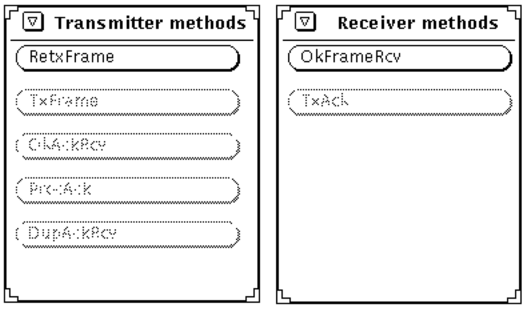


Fig. 57: Result of a one step evolution

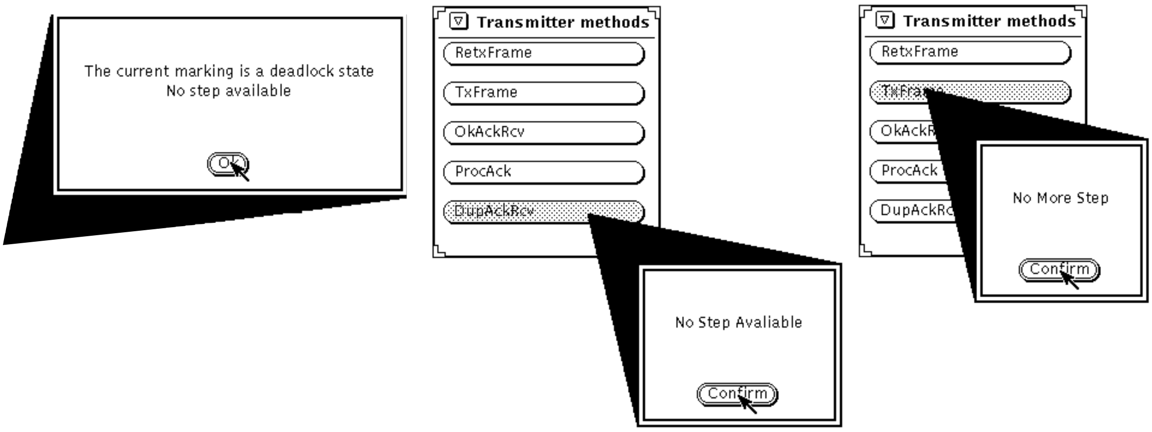


Fig. 58: Messages associated to a *test firable* simulation mode

6.1.7 Transition monitor window

The function of this window (see Figure 60) depends on the environment's context. However, it is roughly used to inspect a list of transitions displayed in the scrolling list **List of transitions**. The selection of an item of this list triggers off the display of the selected transition with its parameters in the field **Transition**, the number of its input and output markings in the field **Source marking** and **Target marking**. Object input and output markings are specified in the two *text windows* in the lower part of the *Transition monitor* window.

When no simulation is under process the markings graph can be explored. The list of transitions shown in the scrolling list **List of transitions** depends on the settings done in the menu **Show transitions** (see Figure 59). In this menu *Selected marking* refers to the number of the marking specified in the numeric field **Selected marking** (see Figure 60).

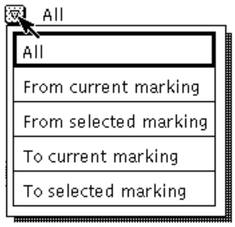


Fig. 59: Menu associated to **Show transitions**

When the simulation mode is on with the **Successor choice** field set to **user defined** (see Figure 55), the list of transitions in the scrolling list **List of transitions** does not depend on the setting of the field choice **Show transition**. In this context, the selection of a button associated to an object's method (see 6.1.6) triggers off the display of all successor markings of the current marking that can be reached with the selected method and its possible parameter instantiations. The successor marking is the target marking of the current transition selected in the scrolling list of transitions. To do so the user also has to click on the button **Successor marking select** (see Figure 53). A message is displayed when the current marking is a deadlock state w.r.t. the set of selected methods (see Figure 58).

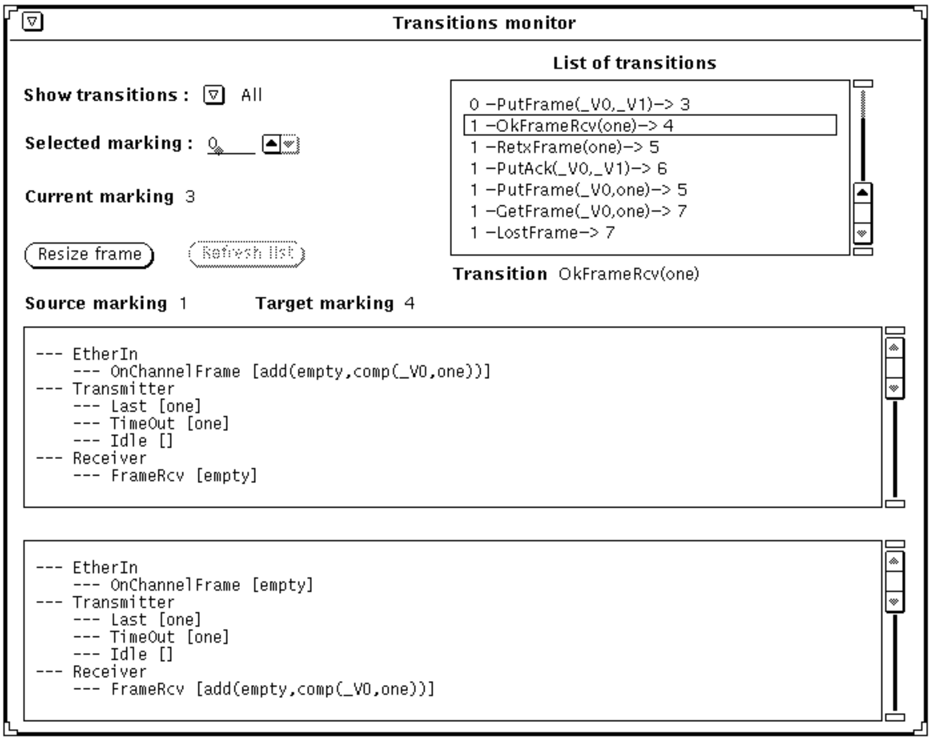


Fig. 60: Transition monitor window

If the **Successor choice** field is set to **randomly**, the transition list is useless in the choice of a successor. The markings graph can therefore be consulted freely. In this mode the scrolling list **List of transitions** is not automatically refreshed, the user has to click on the button Refresh list .

The button Resize frame hides or shows the lower part of the *Transition monitor* window (i.e.

its two text windows).

6.2 Algebraic specification part

During simulation the user can also test the behaviour of abstract data type specifications. This part of the tool allows definitions of free and 'ground' variables, defined as short-cuts of term expressions. The user can evaluate a specification expression where variables may occur. Equality between two expressions can also be proved using resolution techniques. Free variables in expressions are instantiated if needed.

6.2.1 Variable management window

This window (Figure 61) is used to define associations between the name of a variable (field **Var name**) and an algebraic term (**Var expression**). Free variables can also be defined. The expression *Name_of _the_sort* in the field **Var expression** defines a free variable of sort *Name_of_the_sort*. The button (Insert) creates the association between a variable and its definition. The name of a variable is unique. The button (Change) and (Delete) are relative to the current item of the scrolling list **Variables**, (Change) is used to change the variable's name associated to an expression or to change the expression associated to a variable. In the first case the new name cannot be an already defined name. The definition of a variable expression can itself use other variable definitions and should take a cyclic definition.

```
Ex:
            Var name: dummyzero          Var expression: inc(one)
            Var name: verydummyone       Var expression: inc(dummyzero)
```

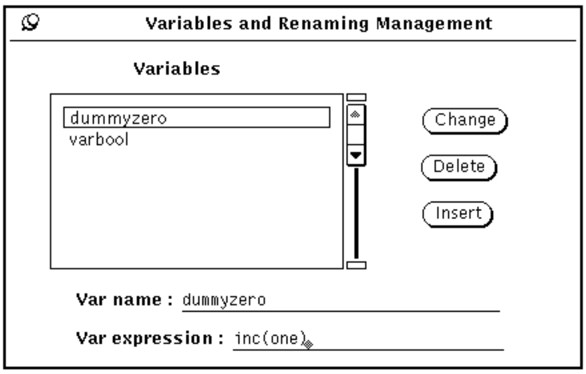


Fig. 61: Window associated to (Variables...)

6.2.2 Algebraic expression window

This window (Figure 62) can be used to test the behaviour of algebraic specifications. Evaluation of terms or equality between two terms can be executed. When free variables appear in terms they are instantiated, as loose as possible, if needed. Due to the overloading and the use of variables, an expression can yield different evaluations. The field **Bound** limits the number of evaluations while the field **Evaluation** specifies the number of the currently displayed evaluation and the field **Number of evaluations** gives the number of evaluations found (maximum up to the bound's limit). The field **Sort** specifies the sort's name of the current evaluation and the field **Used in module** gives the name of the module where this evaluation can be used.

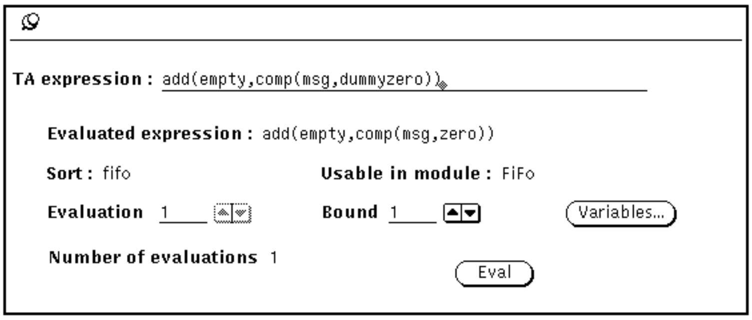


Fig. 62: Window associated to Evaluation ▽ CO-OPN... TA...

The button Eval starts the computation of the expression specified in the field **TA expression**. The button Variables... calls the *Variable management* window (see Figure 62).

6.2.3 CO-OPN Workspace Management window

The user can at any moment save (Save) its current environment or load another (Load). The safeguard includes the set of associations between variables and expressions, the markings graph which include the transitions with their parameters and the place arrangements of the windows. The button Reset allows the user to restart his simulation from the initial environment, all informations about markings and transitions are removed except for the initial marking.

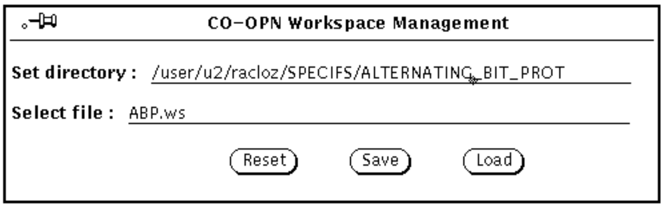


Fig. 63: Window associated to Workspace...

Some simple tests are done when a new environment is loaded to ensure that it corresponds to the system currently under simulation (see Figure 64).

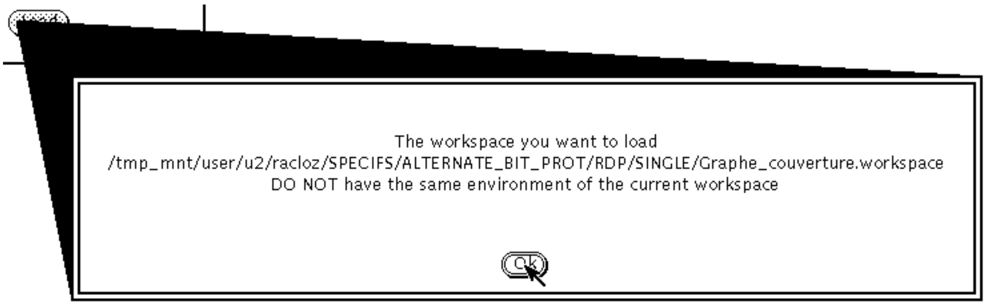


Fig. 64: Message associated to a bad environment loading

6.3 Temporal logic part

6.3.1 Introduction

Branching temporal logic (BTL) provides a formal framework for describing and reasoning about correct behaviour of parallel systems by expressing properties related to the occurrences of events in the time. The operators and modalities of the BTL can be used to express important properties such as invariance, eventuality and precedence.

Given a temporal logic formula expressing a correctness property, the method of model checking is a technique used to verify the truth of the specified property in the structure representing the parallel system. This technique can only be applied to a system with a finite set of states. Symbolic model checking overcomes this limitation by using the notions of predicates and predicate transformers already developed in the literature to the field of temporal logic. A predicate is used to represent finitely an infinite set and the effects of simple predicate transformers (functions) can be linked to the temporal logic operator semantics.

6.3.2 Predicate Manipulation window

A predicate is a set of predicate structures. A predicate structure is composed of a ‘negative’ and ‘positive’ set of markings corresponding respectively to the markings which must and must not belong to the underlying set of markings of the predicate. Definition and calculation of predicates are handled in the *Predicate Manipulation* window (Figure 65).

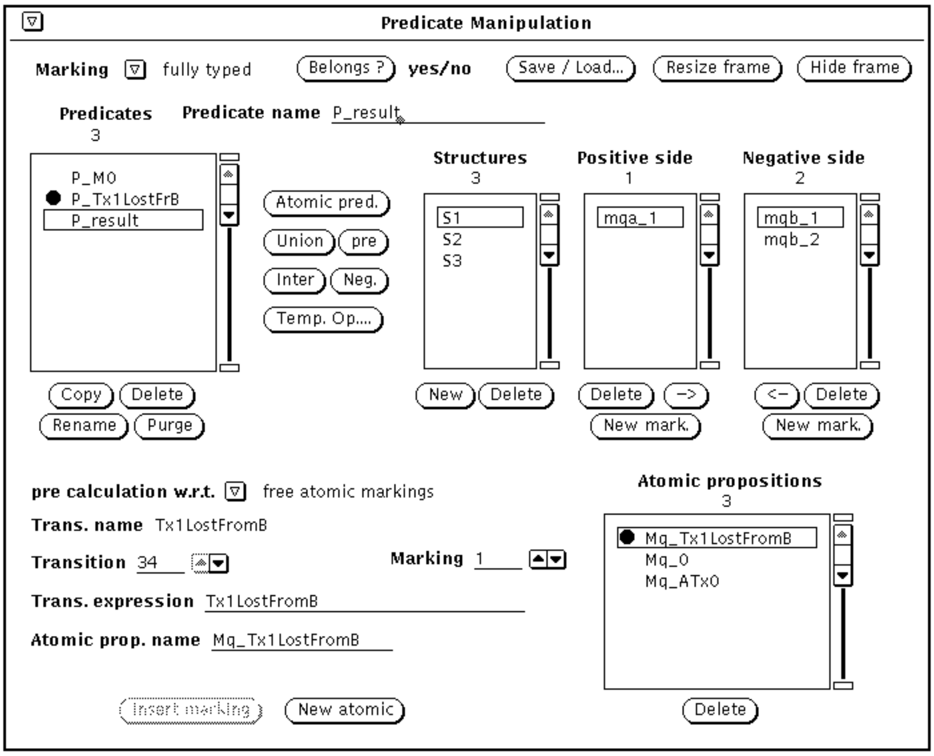


Fig. 65: Window associated to Predicates... of *CO-OPN Monitor*

Conventions

The scrolling lists defined have the following properties. A single item of a list can be surrounded by a rectangle at once, this item will be called the *current item*. The set of items preceded by a black dot are called *selected items*.

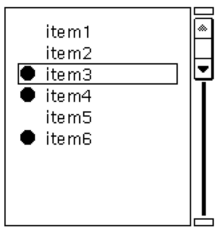


Fig. 66: Example of a scrolling list of items

In Figure 66 the current item is *item3* and the selected items are contained in the set {*item3*, *item4*, *item6*}. The current item does not necessarily belong to the selected items.

Atomic propositions

The list which appears under **Atomic Propositions** (see Figure 65) is a list of names of markings. The user has two possibilities to choose a marking. He can choose a marking which belongs to the set of markings of the current graph of the simulation (*system marking*) or he can choose the *input marking* of a transition of the system. The name associated to the marking is specified in **Atomic prop. name**. The system always proposes a default name that the user can change. When an atomic proposition is created its associated marking is displayed in the *marking window* (see Figure 48).

- System marking

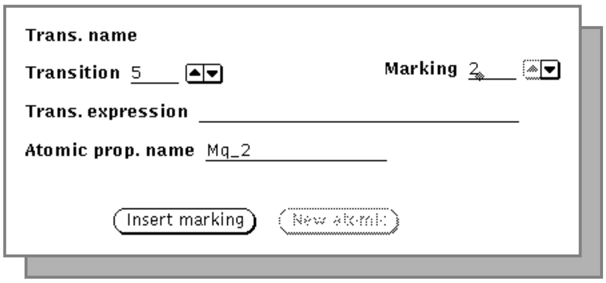


Fig. 67: Selection of a *system marking*

To be able to choose a system marking, the user has to select a marking number in the numeric field **Marking**. If the user clicks on the button (Insert marking) , a new atomic proposition is created and inserted in the scrolling list **Atomic proposition**.

- Input marking

If the user wants to choose an input marking transition, he has to select a transition number in the numeric field **Transition**. The corresponding transition name appears in the field **Trans. name**.

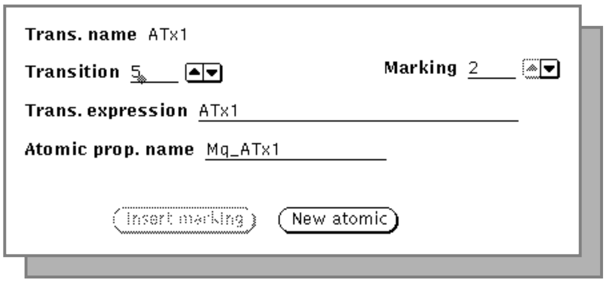


Fig. 68: Selection of an *input markings*

When the selected transition has parameters, the user specifies them in the field **Trans. expression**. A free parameter in the transition is represented by a name which has to start with the underscore character '_'. When the user's choice is made, he clicks on button (New atomic) . Some verifications about the transition parameters such as their types and numbers are performed before creating a new atomic proposition. If a problem is detected the user is informed (see Figure 69).

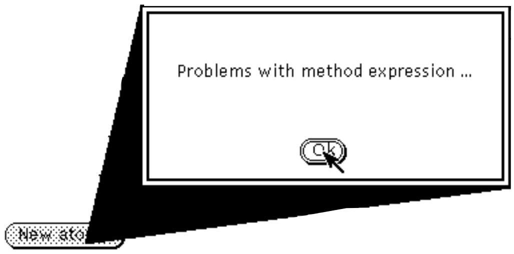


Fig. 69: Transition expression error

The button Delete removes the current atomic proposition of the scrolling list **Atomic propositions** (see Figure 65).

Predicate definition

The button Atomic pred. (see Figure 65) creates a predicate composed of a single predicate structure. This structure has only one marking in its positive set of markings and no marking in its negative set of markings. The single marking is the current atomic proposition of the scrolling list **Atomic propositions**.

Predicate consultation

Structures displayed in the **Structures** scrolling list are those of the current predicate of the **Predicates** scrolling list. Similarly the markings of the scrolling list **Positive side** and **Negative side** are those of the current structure of the scrolling list **Structures**. The selection of a marking of one the scrolling lists **Positive side** or **Negative side** triggers off the display of its contents in the *marking window* (see Figure 48).

Operations on predicates

- Basic operations

These operations are relative to the current predicate of the scrolling list **Predicates**.

- Copy create a new predicate which is a copy of the current predicate and whose name is preceded by '*copy_of_*';
- Delete remove from the scrolling list of **Predicates** its current predicate;
- Rename change the name of the current predicate to the name specified in the field **Predicate name**;
- Purge removes from the current predicate all unsatisfied or redundant structures and removes from the remaining structures their redundant markings (ref. future rapport intern).

- Advanced operations

These operations are relative to the set of selected predicates of the scrolling list **Predicates**.

Unary operations

- (Neg.) calculates the negation of the single selected predicate. The result is specified in the field **Predicate name**.

- (pre) calculates the predicate structure whose underlying set of markings are predecessors of the underlying set of markings of the single selected predicate. A marking is a predecessor of another marking if they are linked by a transition. The user can specify the set of transitions which are to be considered by means of the 'choice field' **pre calculation w.r.t.** (see Figure 70).

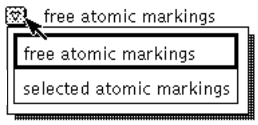


Fig. 70: Menu associated to **pre calculation w.r.t.**

- '**pre calculation w.r.t.** *free atomic markings*'

With this choice the *pre* operation is performed w.r.t. all the transitions defined in the system.

- '**pre calculation w.r.t.** *selected markings*'

When this choice is selected the scrolling list of **Atomic propositions** are refreshed in order to display the list of markings which are defined as an *input marking* of a transition, the markings defined as *system markings* are hidden (see Figure 71). The set of selected atomic propositions can then be linked to the corresponding set of transitions; a transition belongs to this set if its input marking is one of the *input marking*s of the scrolling list **Atomic proposition**. With this choice of **pre calculation w.r.t.** the *pre* operation is performed w.r.t. this set of transitions.

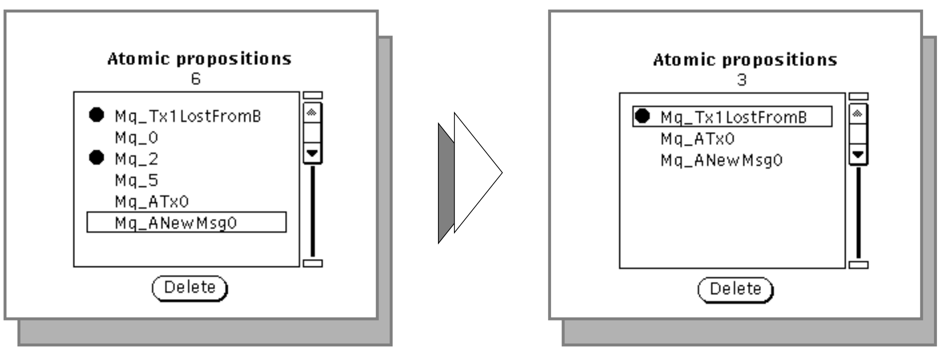


Fig. 71: Effect of the *selected markings* choice

N-ary operations

- The buttons Union and Inter calculate respectively the union and intersection of the selected predicates of the scrolling list **Predicates**. The result is specified in the field **Predicate name.**

Other features

Button Belongs ?

This button allows to test if the current marking belongs to the single selected predicate. The answer is shown beside the button by a **Yes** or **No** message (see Figure 72).

Fig. 72: Belongs message

Button Save / Load...

This button can save or load a set of predicates and remove the set of predicates defined.

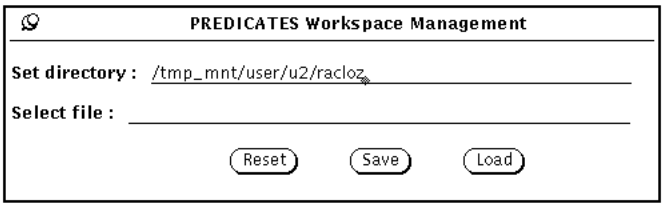


Fig. 73: Window associated to Save / Load...

The button Save saves the set of predicates defined by the user in the specified file. This file is composed of two parts. One is the internal representation of the predicates and the other is written in such way that the user can easily read it.

The structure of a file which includes a saved predicate environment can be described by the following rules of a pseudo-grammar:

```
file →        internal_representation
              readable_representation

internal_representation →< predicate_descr_inter , >+
predicate_descr_inter →
              predicate_struct_NEW ( identifier , integer )
              < predicate_descr_inter_pos >+
              < predicate_descr_inter_neg >+
predicate_descr_inter_pos →
```

predicate_struct_pos_NEW
(identifier , integer , **[** listmarkings **])**

predicate_descr_inter_neg ⟶
predicate_struct_neg_NEW
(identifier , integer , **[** listmarkings **])**

listmarkings ⟶ < marking , >*

marking ⟶ <terme , >$^+$

readable_representation ⟶ < predicate_descr_read , >$^+$

predicate_descr_read ⟶
PREDICATE identifier (*predicate name*)
Nb Struct : integer (*number os structures*)
<structure_descr_read>$^+$

structure_descr_read ⟶
STRUCTURE interger (*structure's number*)
List of markings POSITIVE SIDE
listobjectsmarkings
List of markings NEGATIVE SIDE
listobjectsmarkings

listobjectsmarkings ⟶
identifier (*object's name*)
< --- identifier marking >* (*place's name and its content*)

```
predicate_struct_NEW(P_MqATx0,1).
predicate_struct_pos_NEW(P_MqATx0,0,[[0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]).
predicate_struct_neg_NEW(P_MqATx0,0,nil).
predicate_struct_NEW(P_M0,1).
predicate_struct_pos_NEW(P_M0,0,[[1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0]]).
predicate_struct_neg_NEW(P_M0,0,nil).


{* ---

PREDICATE P_MqATx0
Nb Struct : 1

 STRUCTURE 0
       List of markings POSITIVE SIDE

                  Marking num 0
                  --- ARTS0 (1)

       List of markings NEGATIVE SIDE

PREDICATE P_M0
Nb Struct : 1

 STRUCTURE 0

       List of markings POSITIVE SIDE

                  Marking num 0
                  --- BWait0 (1)
                  --- ARcvWaited1 (1)

       List of markings NEGATIVE SIDE


--- *}
```

Fig. 74: Example of a saved file associated to a simple Petri net

The button Load loads all the saved predicates of the specified file. Already defined predicates of the current environment are kept after the loading except if they are redefined in the file. The predicate name is the key to its identification.

- Reset eliminates all the defined predicates of the current environment;

- -> , <- allow respectively the move of a marking belonging to the set of positive markings toward the set of negative markings and conversely;

- New mark. under the scrolling lists **Positive side** and **Negative side** insert the current marking of the scrolling list in the set of the system markings.

- **Marking**

  The choice made in the field **Marking** (see Figure 75) affects the way terms of the marking of the objects are written in the *method window* (Figure 48). When set to *abbreviated* a term of the form *nameofthemodule_nameoftheterm* is simply written *nameoftheterm*.

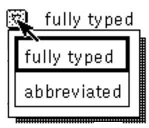


Fig. 75: Menu associated to **Marking**

Fig. 76: Effect of the selection of *Abbreviated*

6.3.3 Temporal Operators window

The Temporal Operators window is called by clicking the Temp. Op.... of the *Predicate Manipulation* window. The result of an operation involving temporal operators is specified in the field **Predicate name** of the *Predicate Manipulation* window.

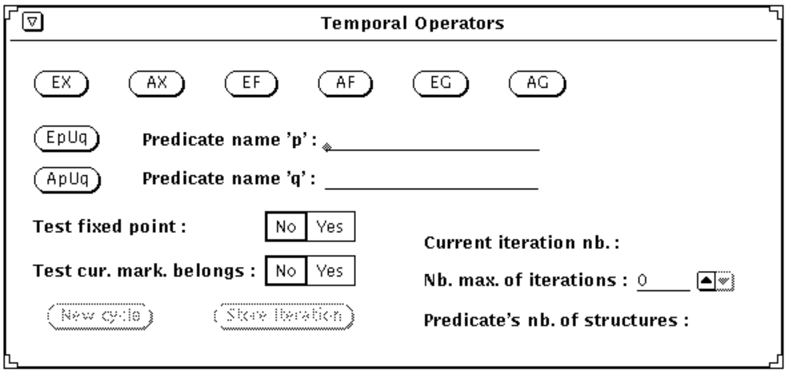


Fig. 77: Window associated to Temp. Op....

The calculation of the effect of a unary operator is performed on the single selected predicate of the scrolling list **Predicates**. These operators are **EX**, **AX**, **EF**, **AF**, **EG** and **AG** and are associated to their corresponding buttons (see Figure 77).

The binary operators are **Ep**Uq and **Ap**Uq. The name of the predicate '*p*' and '*q*' are respectively specified in the field **Predicate name 'p'** and **Predicate name 'q'**.

When an operator involves fixpoint iterations, the user can define an upper bound limit of the number of iterations in the field **Nb. max. of iterations**. If set to 0 this means that no limit is put. When the maximum number of iterations is reached the user can perform a new iteration by clicking on the button New cycle or he can store the current resulting predicate of the last iteration by clicking on the button Store iteration . The test of fixpoint reaching can be set to on or off in the field **Test fixed point**.

When the user is interested in testing whether the current marking of the system belongs to the predicate which is currently calculated he has to set the field **Test cur. mark. belongs** to **Yes**. The user can also set to on or off the test involving the current mark reaching (**Test fixed point:** field).

Bibliography

[1] Buchs D., Guelfi N.
"A Concurrent Object Oriented Petri Nets Approach for system specification", 12th International Conference on Application and Theory of Petri Nets, Aahrus Denmark, june 1991,(M. Silva eds.),pp. 432-454 .

[2] Buchs D., Guelfi N.
"Distributed System Specification using CO-OPN", IEEE 3rd Workshop on Future Trends of Distributed Computing Systems, Taipei, April 1992,pp 26-33.

[3] Buchs D., Guelfi N.
"A semantic Description of Actor Languages using CO-OPN", abstract in 14th International Conference of Software Engineering, Melbourne, Australia, may 1992.

[4] Buchs D., Flumet J.,Racloz P.,
"Producing Prototypes from CO-OPN specifications", IEEE International Workshop on Rapid System Prototyping, June 1992, Research Triangle Park, N. C. pp. 77-93.

[5] Buchs D., Guelfi N.
"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, France september 1992, pp. 261-279 .

[6] Buchs D., Guelfi N.
"Formal Development of Actor Programs using Structured Algebraic Petri Nets", Int. Conference on parallel Architectures and Languages Europe PARLE'93, Munich, LNCS 694, pp 353-366, june 1993.

[7] Buchs D., Flumet J.,Racloz P.,
"SANDS: Structured Algebraic Net Development System", 14th International Conference on Application and Theory of Petri Nets, Tool presentations (Ugo Buy ed) pp 25-29., Chicago U.S.A., june 1993.

[8] Buchs D., Mourlin F., Safavi L.
"CO-OPN for specifying a distributed termination detection algorithm", World Transputer Congress in Aachen, Germany, Workshop on "Software Engineering for Parallel Systems", 20 -24 September, 1993

[9] Buchs D.
"Proof Reduction of Equivalence of Concurrent Systems: A Theory and an Example with the Algebraic Petri Nets", Cahiers du CUI, no 56, Université de Geneve,1992.

[10] Buchs D.
"Observational Equivalence of Algebraic Petri Nets",Cahiers du CUI, no 57, Université de Geneve,1992.

[11] Buchs D., Guelfi N.
"High Level Nets", Cahiers du CUI, no 58, Université de Geneve,1992.

[12] Racloz P., Buchs D.
"Symbolic Proof of CTL Formulae over Algebraic Nets",Cahiers du CUI, no 65, Université de Geneve,1992.

[13] Buchs D., Guelfi N.
"CO-OPN: A Concurrent Object Oriented Petri Nets Model", rapport de recherche, no 616, L.R.I. december 1990, 41 pages.

[14] Buchs D., Guelfi N.
"A semantic Description of Actor Languages by Structured Algebraic Petri Net", rapport de recherche, no 639, L.R.I. january 1991, 21 pages.

[15] Buchs D., Guelfi N.
"Open Distributed Programming using the object oriented specification formalism CO-OPN" rapport de recherche, no 700, L.R.I. october 1991, 22 pages.

[16] Buchs D., Guelfi N.
"System Specification using CO-OPN" rapport de recherche, no 825, L.R.I. march 1993, 46 pages.

[17] Buchs D., Brunet D.,Mourlin M
"Modular Asynchronous Visual Information processor", Cahiers du CUI, no ?, Université de Geneve,1993

[18] Racloz P., Buchs D.
"Symbolic Proof of CTL Formulae over Petri Nets",Cahiers du CUI, no 76, Université de Geneve,1993.

[19] Racloz P., Buchs D.
"Verification of Temporal Properties on an Algebraic Modelling: The Alternating Bit Protocol", Cahiers du CUI, no ?, Université de Geneve,1993.

Annex 1.
Alternating Bit Protocol algebraic specification

spec Bool;
sorts bool;
generated by
true : -> bool;
false : -> bool;
end

spec Message;
sorts message;
end

spec AltBit;
use Bool;
sorts altbit;
generated by
0 : -> altbit;
1 : -> altbit;
operations
inc : altbit -> altbit ; %modulo 2%
axioms
i1: => inc(0) = 1;
i2: => inc(1) = 0;
end

spec Frame;
use Message, AltBit;
sorts frame;
generated by
comp : message, altbit -> frame;
end

spec FiFo;
use Frame, Bool, AltBit, Message;
sorts fifo;
generated by
empty : -> fifo;
add : fifo, frame -> fifo;
operations
is-empty : fifo -> bool;
rm : fifo -> fifo;
get : fifo -> frame;
numframe : fifo -> altbit;
axioms
e1 : => is-empty(empty) = true;
e2 : => is-empty(add(ff,fr)) = false;
r1 : => rm(empty) = empty;
r2 : => rm(add(empty,fr)) = empty;
r3 : is-empty(ff) = false => rm(add(ff,fr)) = add(rm(ff),fr);
g1: => get(add(empty,fr)) = fr;
g2: is-empty(ff) = false => get(add(ff,fr)) = get(ff);
n1: is-empty(ff) = false => numframe(add(ff,fr)) = numframe(ff);
n2: => numframe(add(empty,comp(msg,num))) = num;
where
ff : fifo;
fr : frame;
msg : message;
num : altbit;
end

Annex 2.
CO-OPN objects

Remarks:

- Description of CO-OPN objects has been automatically generated from their graphical specifications;
- The '%' symbols are comment delimiters.

ReceiveXX

object ReceiveXX;

use
FiFo % (fifo) % ,
Frame % (frame) % ,
AltBit % (altbit) % ,
Message % (message) % ,
EtherOut % Synchro depuis (TxAck) %,
EtherIn % Synchro depuis (OkFrameRcv) % ;

places
FrameRcv : [fifo];

methods
TxAck : altbit ;
OkFrameRcv : altbit ;

axioms
a396:TxAck(b) with PutAck(msg, b) ::
=> FrameRcv([add(ffrcv,comp(msg,b))])
-> FrameRcv([ffrcv]) ;

a394:OkFrameRcv(b) with GetFrame(msg, b) ::
=> FrameRcv([ffrcv])
-> FrameRcv([add(ffrcv,comp(msg,b))]) ;

marking
FrameRcv([empty]);

where
msg : message;
ffrcv : fifo;
b : altbit;

end

Transmitter

object Transmitter;

use
FiFo % (fifo) % ,
Frame % (frame) % ,
AltBit % (altbit) % ,
Message % (message) % ,
EtherOut % Synchro depuis (DupAckRcv OkAckRcv,) %,
EtherIn % Synchro depuis (TxFrame RetxFrame,) % ;

places
Idle : [altbit];
TimeOut : [altbit];
AckRcv : [altbit];
Last : [altbit];

methods
DupAckRcv : altbit ;
ProcAck ;
OkAckRcv ;
TxFrame : altbit ;
RetxFrame : altbit ;

axioms
a379:DupAckRcv(frnum) with GetAck(msg, frnum) ::
acknum = inc(frnum)
=> Last([acknum])
-> Last([frnum]) ;

a301:ProcAck::
=> AckRcv([acknum]), TimeOut([frnum])
-> Idle([acknum]) ;

a300:OkAckRcv with GetAck(msg, frnum) ::
=> Last([frnum])
-> AckRcv([frnum]), Last([inc(frnum)]) ;

a302:TxFrame(frnum) with PutFrame(msg, inc(frnum)) ::
=> Idle([frnum])
-> TimeOut([inc(frnum)]) ;

a372:RetxFrame(frnum) with PutFrame(msg, frnum) ::
=> TimeOut([frnum])
-> TimeOut([frnum]) ;

marking
Idle([zero]),
Last([one]) ;%init%

where
msg : message;
acknum : altbit;
frnum : altbit;

end

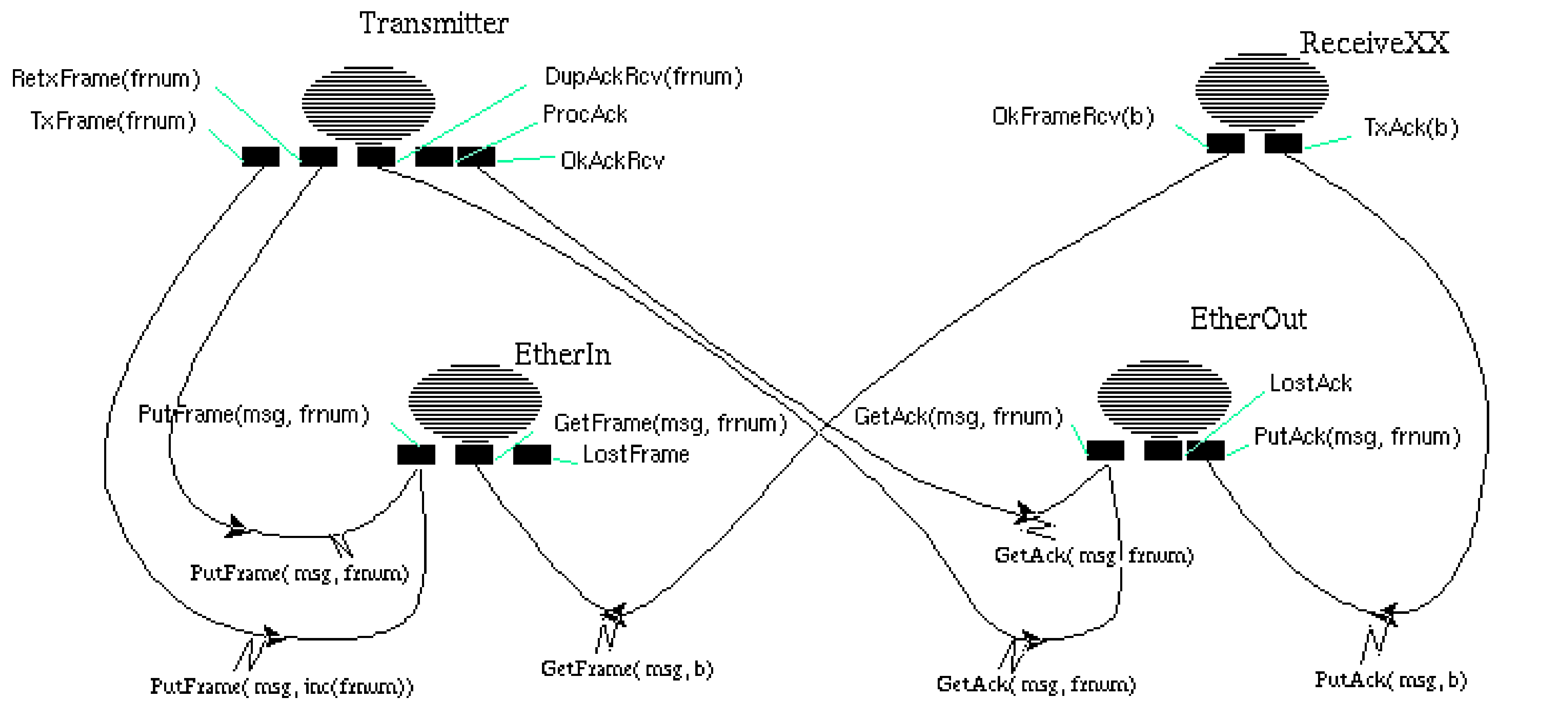
EtherIn

```
object EtherIn;

use
 FiFo % (fifo) % ,
 Frame % (frame) % ,
 AltBit % (altbit) % ,
 Message % (message) % ;

places
 OnChannelFrame : [fifo];

methods
 PutFrame : message, altbit ;
 GetFrame : message, altbit ;
 LostFrame ;

axioms
 a318:PutFrame(msg, frnum)::
=> OnChannelFrame([ffch])
 -> OnChannelFrame([add(ffch,comp(msg,frnum))]) ;

 a316:GetFrame(msg, frnum)::
=> OnChannelFrame([add(ffch,comp(msg,frnum))])
 -> OnChannelFrame([rm(add(ffch,comp(msg,frnum)))]) ;

 a314:LostFrame::
=> OnChannelFrame([ffch])
 -> OnChannelFrame([rm(ffch)]) ;

marking
 OnChannelFrame([empty]) ;%init%

where
 ffch : fifo;
 frnum : altbit;
 msg : message;

end
```

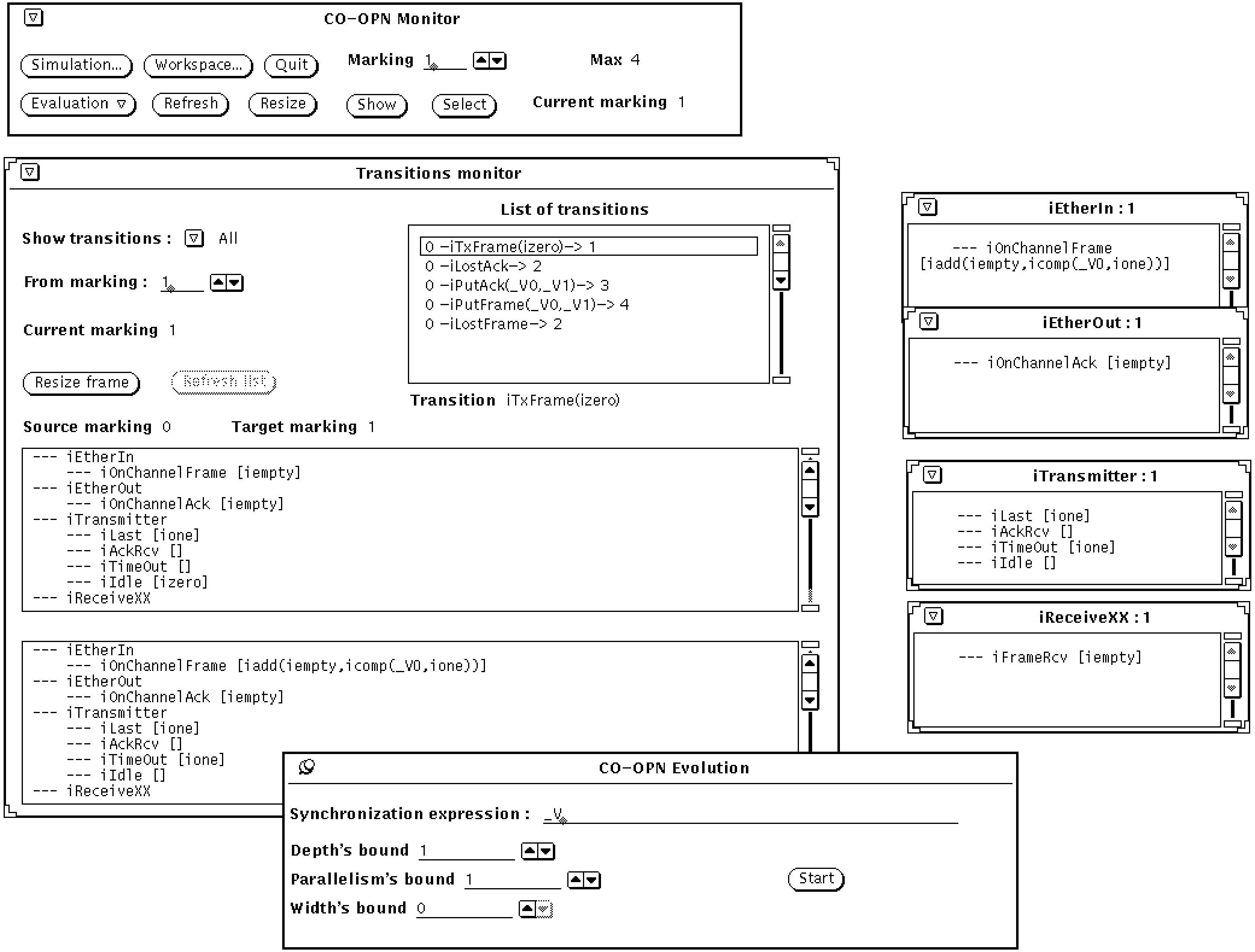
EtherOut

```
object EtherOut;

use
 FiFo % (fifo) % ,
 Frame % (frame) % ,
 AltBit % (altbit) % ,
 Message % (message) % ;

places
 OnChannelAck : [fifo];

methods
 GetAck : message, altbit ;
 LostAck ;
 PutAck : message, altbit ;

axioms
 a327:GetAck(msg, frnum)::
=> OnChannelAck([add(ffch,comp(msg,frnum))])
 -> OnChannelAck([rm(add(ffch,comp(msg,frnum)))]) ;

 a325:LostAck::
=> OnChannelAck([ffch])
 -> OnChannelAck([rm(ffch)]) ;

 a323:PutAck(msg, frnum)::
=> OnChannelAck([ffch])
 -> OnChannelAck([add(ffch,comp(msg,frnum))]) ;

marking
 OnChannelAck([empty]) ;%init%

where
 fr : frame;
 ffch : fifo;
 frnum : altbit;
 msg : message;

end
```

Annex 3.
Global view of objects



Annex 4.
Global view of windows



Global view of windows cont.