$\textbf{SANDS}_{1.5}$ / $\textbf{CO-OPN}_{1.5}$

An Overview of the Language and its Supporting Tools

O. Biberstein
M. Buffo
J. Flumet
G. Di Marzo

D. Buchs
C. Buffard
J. Hulaas
P. Racloz

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

LGL-LCO-DI
Swiss Federal Institute of Technology
CH-1015 Lausanne, Switzerland
authorname@di.epfl.ch

November 9, 1995

Technical report #95/133

Abstract

In this document we give an overview of the CO-OPN$_{1.5}$ (Concurrent Object--Oriented Petri Nets) specification language and describe the features of each tool provided in the SANDS$_{1.5}$ (Structured Algebraic Net Development System) development environment.

The CO-OPN$_{1.5}$ language is a specification language devised to support the development of large concurrent systems. The underlying formalisms of the language are algebraic specifications and Petri nets in which tokens correspond to algebraic values. Furthermore, in order to deal with large specifications, some structuring principles have been introduced and in particular, object-orientation paradigm has been adopted for the Petri nets. This means that a CO-OPN$_{1.5}$ specification is a collection of objects which interact concurrently. Interaction between the objects is achieved by means of synchronization expressions which allow the designer to select the object interaction policies.

The development system provides many different tools such as a syntax checker, a simulator, a property verifier based on temporal logic, a graphic editor, a transformation tool supporting the derivation of specifications, an Ada translator which allows to analyze Ada programs in the CO-OPN$_{1.5}$ framework, and a MIMD compiler.

Keywords: Formal methods for software engineering, specification language, modular specifications, high level nets, algebraic Petri nets, algebraic abstract data types, net simulation, reachability graph, model checking, symbolic representation of states, verification, temporal properties, high performance, parallelism, probabilism, specification transformation, prototyping, executable specifications, distributed systems.

Contents

1 SANDS$_{1.5}$ / CO-OPN$_{1.5}$

Olivier Biberstein & Didier Buchs

1.1 Introduction

The CO-OPN [BG91] (Concurrent Object-Oriented Petri Nets) project was begun in 1991 with the aim of providing a specification language and a set of tools for the design of large concurrent systems. Some tools were packaged two years later in the SANDS [BFR93a] (Structured Algebraic Net Development System) development environment.

Recently, a new version of the language, called CO-OPN$_{1.5}$, has been completed, offering several improvements, and some new tools have also been developed. These new features of the language and the additional tools have naturally led to develop a fresh development system called SANDS$_{1.5}$.

The goal of this document is to give an overview of the CO-OPN$_{1.5}$ specification language and to briefly introduce each new tool which accompanies the development system SANDS$_{1.5}$. Moreover, several examples are provided in this document which illustrate the characteristics of the language.

1.1.1 The CO-OPN$_{1.5}$ Language

The CO-OPN$_{1.5}$ language is a specification language based on both algebraic specifications and algebraic Petri nets formalisms [Rei91]. The former formalism stands for data structures aspects while the latter stands for behavioral and concurrent aspects of the systems. These two formalisms are quite useless when large problems are confronted and some structuring capabilities are needed to overcome their limitations. Thus, for the underlying formalism we have adopted the object-oriented paradigm, which means that a CO-OPN$_{1.5}$ specification is a collection of objects which interact concurrently. Not all the object-oriented notions are present in the CO-OPN$_{1.5}$ language. For instance the class notion is missing and in this sense the language is said to be object-based. Yet another project, called CO-OPN/2 [BB95] is under way with the objective of integrating the missing object-oriented notions.

1.1.2 The SANDS$_{1.5}$ Development Environment

The SANDS$_{1.5}$ development system consists of several tools including

- a graphic editor, which can be used for graphically describing a specification,
- a checker, which verifies the consistency of a textual part of a CO-OPN$_{1.5}$ specification,
- a simulator, which allows to simulate the Petri/Algebraic nets that compose a specification,
- a verifier, which is able to prove some temporal properties of a specification,
- a Ada translator, which translates Ada programs into CO-OPN$_{1.5}$ and allows one to analyze such programs in the CO-OPN$_{1.5}$ framework,

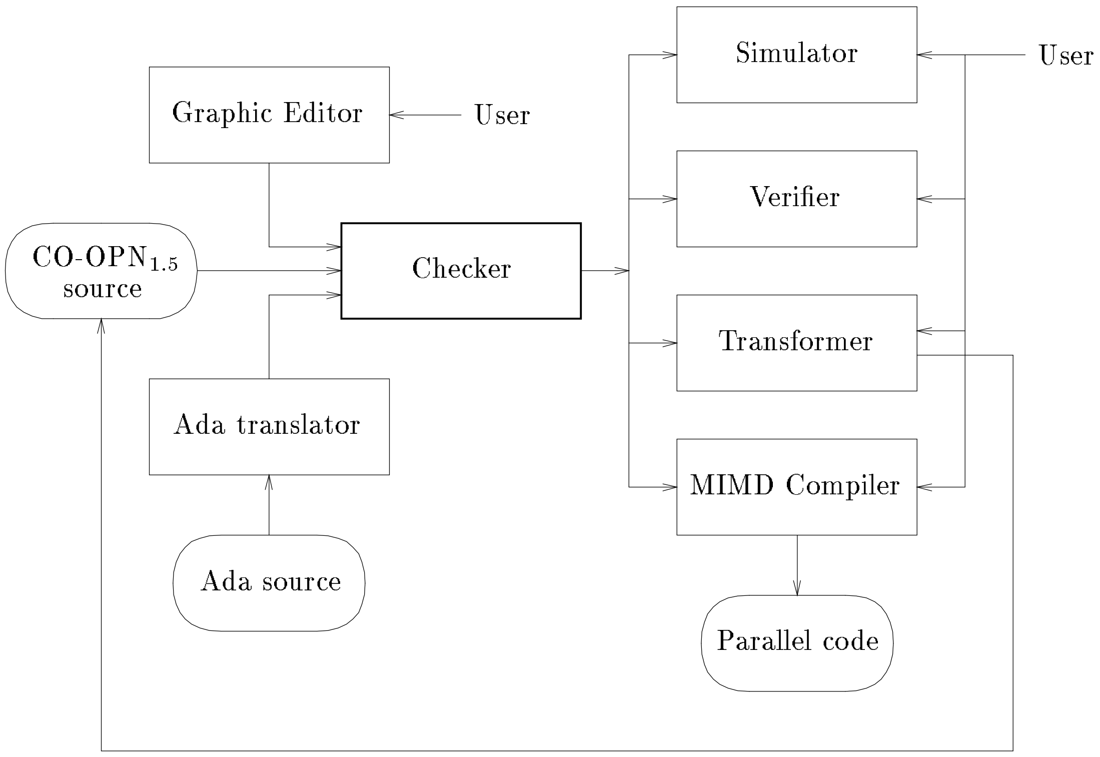


Figure 1.1: An overview of the SANDS$_{1.5}$ development system.

- a transformation tool, which can be used, with its accompanying library, for transforming an abstract specification into a more concrete one,
- and a MIMD compiler, which produces parallel code from concrete CO-OPN$_{1.5}$ specifications.

Figure 1.1 gives an overview of the relationships between the different basic tools composing the core of the SANDS$_{1.5}$ environment. A specification can be written by means of any text editor or by means of the graphic editor of the environment. The Checker is one of the main pieces in this puzzle, it verifies the consistency of a specification written by means of the graphic editor or any text editor and provides all information required by the other tools. The user may interact with every tool or provide them with additional information in order to assist the tools.

1.1.3 Plan of the Document

This document is divided into ten main parts. Part 1, this part, corresponds to a short introduction of the CO-OPN$_{1.5}$ language and the SANDS$_{1.5}$ development environment. Part 2 and 3 are related to CO-OPN$_{1.5}$ while the others are dedicated to a brief description of each tool implicated in SANDS$_{1.5}$. Part 2 introduces the CO-OPN$_{1.5}$ language by means of simple examples while, Part 3 provides several examples which illustrate the features previously introduced. Part 4 to 10 correspond, respectively, to the checker tool, the simulator, the verifier, the transformation tool, the Ada translator and the MIMD compiler. Moreover, two appendixes are provided. The former consists of many standard abstract data types and the latter gives the lexical aspects of the CO-OPN$_{1.5}$ language as well as its BNF-like grammar.

2 CO-OPN$_{1.5}$ Language

Olivier Biberstein

2.1 Introduction

The CO-OPN$_{1.5}$ concurrent specification language is based on both algebraic specifications and algebraic Petri nets formalisms. The former formalism represent the data structures aspects, while the latter stands for the behavioral and concurrent aspects of the systems. In order to deal with large specifications some structuring capabilities have been introduced. The object-oriented paradigm has been adopted, which means that a CO-OPN$_{1.5}$ specification is a collection of objects which interact concurrently. Cooperation between the objects is achieved by means of a synchronization mechanism, i.e. each object event may request to be synchronized with some methods (parameterized events) of one or a group of partners by means of a synchronization expression.

Not all the object-oriented notions are present in the CO-OPN$_{1.5}$ language. For instance, the class notion is missing, and it is in in this sense that the language could be said to be object-based.

A CO-OPN$_{1.5}$ specification consists of a collection of two different modules: the *abstract data type modules* and the *object modules*. The abstract data type modules concern the data structure component of the specifications, and many-sorted algebraic specifications are used when describing these modules. Furthermore, the object modules represent the concept of encapsulated entities that possess an internal state and provide the exterior with various services. For this second sort of modules, an algebraic net formalism has been adopted. Algebraic nets, a kind of high level nets, are a great improvement over the Petri nets, i.e. Petri nets tokens are replaced with data structures which are described by means of algebraic abstract data types. For managing visibility, both abstract data type modules and object modules are composed of an *interface* (which allows some operations to be visible from the outside) and a *body* (which mainly encapsulates the operations properties and some operation which are used for building the model). In the case of the objects modules, the state and the behavior of the objects remain concealed within the body section.

Now we are going to informally describe the syntactic aspects of the CO-OPN$_{1.5}$ language and will use some well-known running examples in order to give an overview of the language. The introduction is composed of two steps. First, the aspects related to the algebraic specifications by means of some classical abstract data types such as the booleans, the natural numbers and a fifo of messages are introduced. In the second phase of the introduction the majority of these abstract data types are employed in the examples related to the object part.

2.2 Abstract Data Types

As has previously been mentioned, a CO-OPN$_{1.5}$ specification consists of a collection of two different kinds of modules: the **`Adt`**[1] modules, which are related to the data structure part of a specification, and the **`Object`** modules, which are dedicated to the dynamic aspects of the system. Both of these modules are composed of an **`Interface`** and a **`Body`** section as well as a third component the module header. In the first basic examples, the header consists of a declaration of the module type (**`Adt`** or **`Object`**), followed by the module name. Nevertheless, when genericity is considered, headers may grow bigger and contain information such as the module type (**`Generic`** or **`Parameter`**), the formal or actual parameter module names as well as the associated correspondences.

On one hand, the **`Interface`** section comprises all the components accessible from the outside. Usually a list of sorts (type names) follows the **`Sorts`** (singular or pural) field, and some operations are defined under both **`Generators`** and **`Operations`** fields. These operations are coupled with their profile. On the other hand, the operation properties expressed by means of equations or axioms remain concealed in the **`Body`** section. These equations lie under the **`Axioms`** field and are established as follows:

$$[\ \mathit{Cond}\ \texttt{=>}\ ]\ \mathit{Expr}_1\ \texttt{=}\ \mathit{Expr}_2\ \texttt{;}$$

Each axiom is an equation which relates two expressions, Expr_1 Expr_2, which are constructed from the module interface, and states that both expressions denote the same value(s). An optional condition, *Cond*, may be added, determining the context in which an axiom holds true. Some auxiliary properties may be mentioned under the field **`Theorem`**. From the user point of view, the theorems are logical consequences of the axioms, they may be used for implementation purposes. Of course, variables are available whenever defined under the **`Where`** field.

Figure 2.1 depicts the well-known booleans' and natural numbers' specification. Underneath the two fields **`Operations`** and **`Generators`**, lie the operation profiles in which the underscore character "`_`" gives the position of respective arguments. This *mix-fix* notation allows operations to be defined in a more natural way; in particular, it allows for pre-fix, post-fix, in-fix as well as out-fix (as in `{_}` in singleton set formation).

A useful feature for making specifications easier to understand is *overloading*. Overloading allows one to assign the same identifier to different things. For example, both `Bool` and `Nat` modules of Figure 2.1 define the predicate "`_ = _`" with each having a different profile. Both these predicates refer to equality, but this is to an equality between operands of different sorts.

Since algebraic specifications are divided into modules, a module may need some components from another module. We call these modules, respectively, a *user* module and a *used* module. The user module expresses this kind of dependency by means of the **`Use`** field, followed by the list of all the modules it needs. For example, module `Nat` must declare in its interface a **`Use`** field in which appears the module `Bool`. This is because the operation "`_ = _ : nat nat -> bool`" of module `Nat` deals with the sort `bool` defined in module `Bool`. Thus, a user module is allowed to deal with all the components declared in the used modules' interfaces, while, all the components of the bodies remain inaccessible from the outside. However, the **`Use`** field has two different meanings, depending on where the **`Use`** declaration appears. When the **`Use`** field appears in the interface section, the user module's interface is enriched with the used modules' interfaces. In other words, if a module M_1 uses two modules M_2 and M_3, where M_2 uses M_3, then M_1 does

[1]The Courier bold face words correspond to the reserved words of the language. These reserved words are not case sensitive and can be singular or pural

```
Adt Bool;
Interface
  Sort bool;
  Generators
  true, false : -> bool;
  Operations
  not _   : bool -> bool;
  _ or _  : bool bool -> bool;
  _ and _ : bool bool -> bool;
  _ = _   : bool bool -> bool;
Body
  Axioms
  not true    = false;
  not false   = true;

  true  or x  = true;
  false or x  = x;

  true  and x = x;
  false and x = false;

  (true=true)   = true;
  (true=false)  = false;
  (false=true)  = false;
  (false=false) = true;
  Theorem
  not(x and y) = (not x)or(not y);
  Where
  x, y : bool;
End Bool;
```

```
Adt Nat;
Interface
  Use Bool;
  Sort nat;
  Generators
  0 : -> nat;
  succ _ : nat -> nat;
  Operations
  _ + _  : nat nat -> nat;
  _ = _  : nat nat -> bool;
;;  some other operations
Body
  Axioms
  0+n = n;
  (succ n)+m = succ(n+m);

  0=0 = true;
  (0=succ n) = false;
  (succ n=0) = false;
  (succ n=succ m) = n=m;
;;  and their associated axioms
  Where
  n, m : nat;
End Nat;
```

Figure 2.1: The Classical Booleans' and Natural Numbers' Abstract Data Types

not need to declare the use of M_3. This property does not hold when the **Use** field declaration appears in the body section.

Genericity is another way of making specifications more concise without sacrificing their legibility. Generic abstract data types are data structures which depends on some formal parameter modules. The header of a generic data type must begin with the **Generic** keyword, followed by the module name and the list of its formal parameter modules. An example of a generic first-in-first-out data type is illustrated in Figure 2.2. Module `Elem` corresponds to the formal parameter module, which is replaced by an actual module during the instantiation process. The `Elem` module must be defined as a **Parameter** module as is shown in Figure 2.2. Various information can be defined in a parameter module as we will see. However, information as to the sort definition is essential because this specific sort is used later in the generic module. Note that the implicit use declaration of `Elem` in the `Fifo` module and the mix-fix notation in the operations "`add _ to _`" and "`empty? _`". Moreover, the absence of axioms related to the operations "`next empty`" and "`remove empty`" implies an undefined value.

The instantiation mechanism has to be invoked in order to obtain an actual module from a generic module, where the formal parameter module is replaced those which are actual. This is accomplished by defining a new module which specify the generic module, all the actual parameter modules names, the correspondences between the sorts of the formal parameter modules and those of the actual parameter modules, as well as a renaming of some identifiers. This mechanism can

```
Generic Adt Fifo(Elem);
Interface
  Use Bool;
  Sort fifo;
  Generators
  empty      : -> fifo;
  add _ to _ : elem fifo -> fifo;
  Operations
  empty? _   : fifo -> bool;
  next _     : fifo -> elem;
  remove _   : fifo -> fifo;
Body
  Axioms
  empty? empty = true;
  empty? add e to f = false;

  remove add e to empty = empty;
  remove add e to add e' to f =
    add e to remove add e' to f;

  next add e to empty = e;
  next add e to add e' to f =
    next add e' to f;
  Where
  f : fifo;  e,e' : elem;
End Fifo;
```

```
Parameter Adt Elem;
Interface
  Sort elem;
Body
End Elem;
```

Figure 2.2: The Generic First-In-First-Out Adt and its Parameter Module.

be performed when the actual parameter modules satisfy the properties of the formal parameter modules. Figure 2.3 illustrates the instantiation of a the generic module `Fifo` by the module `Nat`, in order to obtain a first-in-first-out data type based on natural numbers. The new keyword **As** means that the new module `FifoNat` comprises all the components of the generic module `Fifo`, where formal parameters are substituted by the module `Nat`. The **Morphism** field provides a correspondence between the sorts `elem` and `nat` while the **Rename** field takes charge of the renaming of the sort `fifo` into the new sort `fifonat`. Here, both these two fields are essential.

```
Adt FifoNat As Fifo(Nat);
Morphism elem -> nat;
Rename fifo -> fifonat;
Interface
Body
End FifoNat;
```

Figure 2.3: A First-In-First-Out Adt of Natural Numbers.

Another example regarding the instantiation of the `Fifo` module is shown in Figure 2.4. Let `Message` be a module for describing some messages. The operations and the messages themselves are irrelevant. Therefore we only define the sort `message` and leave all the other fields empty. The definition of a first-in-first-out data type of messages is realized in the same manner as was previously the case.

```
Adt Message;
Interface
  Sort message;
Body
End Message;
```

```
Adt FifoMessage As Fifo(Message);
Morphism elem -> message;
Rename fifo -> fifomessage;
Interface
Body
End FifoMessage;
```

Figure 2.4: A First-In-First-Out Adt of Messages.

Our last example about genericity is the specification of the generic cartesian product. (See Figure 2.6.) This example illustrates a generic module composed of two different formal parameter modules in which the operation "_ = _" must be defined. Indeed, since the equality operation is defined on both sorts `elem1` and `elem2` in module `CP`, both of the parameter modules must define this operation without mentioning the properties related to the equality of the parameter's data structures. However, since the data structures of the formal parameters are presently unknown, no axiom can be provided. These properties will be provided by the actual parameter. Then, the properties of the equivalence relation of the equality predicate, i.e. reflexivity, transitivity and symmetry, must be expressed by means of the **Theorems** field. (See Figure 2.5.) The `Elem2` module, which is quite similar and defines the `elem2` sort, is not given here. Figure 2.7 gives the instantiation of the `CP` module, in order to obtain the cartesian product based on the boolean and natural sorts.

```
Parameter Adt Elem1;
Interface
  Use Bool;
  Sort elem1;
  Operation
  _ = _ : elem1 elem1 -> bool;
Body
  Theorems                ;; equivalence relation properties
  x=x = true;                                 ;; reflexivity
  (x=y = true) => (y=x = true);               ;; symmetry
  (x=y = true) and (y=z = true) => (x=z = true); ;; transitivity
  Where
  x,y,z : elem1;
End Elem1;
```

Figure 2.5: The Parameter Modules of the Generic Cartesian Product.

2.3 Objects

The previous section introduces the data structure or static aspects (abstract data type module) of the CO-OPN$_{1.5}$ specifications. At this time we may describe the (concurrent) behavioral or dynamic aspects of the specifications. It is however necessary to recall that behavioral aspects are described by means of a second kind of module, known as the object modules. The underlying formalism adopted for these modules is the Algebraic Petri Nets, i.e. Petri nets in which the tokens are abstract data types expressed by means of algebraic specifications. In a similar manner as with the algebraic specifications, some structuring capabilities have been introduced into the algebraic nets formalism

```
Generic Adt CP(Elem1, Elem2);
Interface
  Use Bool;
  Sort cp;
  Generators
  pair : elem1 elem2 -> cp;
  Operations
  fst : cp -> elem1;
  snd : cp -> elem2;
  _ = _ : cp cp -> bool;
Body
  Axioms
  fst (pair x y) = x;
  snd (pair x y) = y;

  (pair x y=pair u v) = (x=u and y=v);
  Theorems                  ;; equivalence relation properties
  a=a = true;                                  ;; reflexivity
  (a=b = true) => (b=a = true);                ;; symmetry
  (a=b = true) and (b=c = true) => (a=c = true); ;; transitivity
  Where
  x,u : elem1;   y,v : elem2;   a, b, c : cp;
End CP;
```

Figure 2.6: The Generic Cartesian Product.

```
Adt CPBoolNat As CP(Bool, Nat);
Morphisms elem1 -> bool;
          elem2 -> nat;
Rename cp -> cpboolnat;
Interface
Body
End CPBoolNat;
```

Figure 2.7: An Instantiation of the Cartesian Product.

in order to cope with large specifications. Thus, the **Object** modules play the important role of encapsulated entities, entities that possess an internal state and provide the exterior with various services. Furthermore, a synchronization mechanism is provided in the furtherance of (concurrent) object interactions. This mechanism is based upon parameterized events, known also as methods, and synchronization expressions. In other words, any object may ask to be synchronized with some of the methods of one or a group of partners and, moreover, it may selects the interaction procedure by means of a synchronization expression.

The object modules possess an **Interface**, a **Body** and a header section, which play the same role as the abstract data types modules but which contain different information. The interface section comprises mainly of the methods (parameterized events) provided by the module, while the state and the operational aspects of the object remain concealed within the body section. Note that no co-domain is provided in the methods' profile. Since these parameterized events take part in a synchronization no value is returned by these events. Nevertheless, information may be exchange during a synchronization. The object state is represented, as usual, in Petri nets by means of a collection of places (i.e. multi-sets of algebraic values). The object state can be modified by means of two kinds of events: the methods (which correspond to the external object stimuli) and the internal transitions (which are hidden in the body and represent the spontaneous object reactions). Sometimes we call these two types of events the observable events (methods) and the invisible events (internal transitions).

The events' effects, data flow and synchronization requests, are described by means of *behavioral axioms* which are established as follows:

[*Cond* =>] *Event* [**with** *Sync*] : *Pre* -> *Post*

in which *Cond* is a condition of the algebraic values of the axiom, *Event* is an internal transition name or a method with parameters and *Sync* is an optional synchronization expression involving some partner's methods and the three operators: "`..`", "`//`" and "`+`" for sequence, simultaneity and non-determinism, respectively. The **With** keyword plays the role of a behavioral abstraction while *Pre* and *Post* correspond, respectively, to what is consumed and to what is produced at the different places within the nets.

In order to explain this second kind of module more precisely, we first present the well-known example of the producer-consumer and then illustrate different synchronization schemes.

Figure 2.8, 2.9 and Figure 2.10 illustrate the specification of a producer and a consumer which exchange some messages through a buffer based on a fifo of messages, as specified on page 6. An outline accompanies each textual representation of the objects, in which the insides of the ellipses include the encapsulated elements (body section), the circles indicate the places, and the arrows represent the data flow. In the case of the rectangles, those which are dark indicate the methods and those which are white correspond to the internal transitions.

```
Object Buffer;
Interface
  Use Message;
  Methods
  get _ , put _ : message;
Body
  Use FifoMessage;
  Place   buffer _ : fifomessage;
  Initial buffer empty;
  Axioms
  put m : buffer b -> buffer (add m to b);

  is b empty? = false =>
  get (next b) : buffer b -> buffer (remove b);
  Where
  m : message;  b : fifomessage;
End Buffer;
```

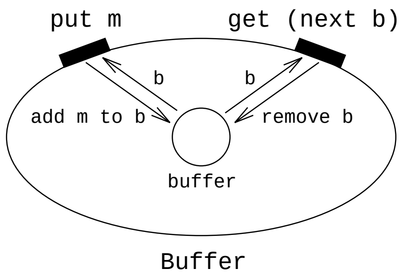


Figure 2.8: The `Buffer` Object, its Textual Representation and its Outline

As shown in Figure 2.8, the `Buffer` object possesses two methods called `put` and `get` which are equipped with a parameter of sort `message` which respectively allows the buffer to import and export a message (field **`Methods`**). The messages will be added and removed from a fifo of messages stored in the place `buffer` mentioned under the field **`Place`**. The **`Initial`** field gives the initial multi-set value for each place with the default value as the empty multi-set. In this case, the initial value of the place `buffer` is an empty fifo of messages. The behavioral axioms are listed under the field **Axioms**, and here they only express the data flow of the methods since no synchronization is requested. The behavioral axiom associated with the method `put` means that a fifo `b` is taken from the `buffer` place, and then the received message `m` is added to `b` which is then put into in the `buffer` place. In other words, the fifo of the buffer obtain one more message. The axiom of the `get` method operates in the same manner except that one message is removed and exported. Note the condition which expresses that the `get` event can only occur when the fifo is not empty. Moreover, because the `FifoMessage` module is not used in the object interface it can be declared in the body section.

```
Object Producer;
Interface
  Method send;
Body
  Use Message, Buffer;
  Place container _ : message;
  Initial
  container m1, container m2, container m3;
  Axioms
  send With put mes : container mes -> ;
  Where
  mes, m1, m2, m3 : message;
End Producer;
```

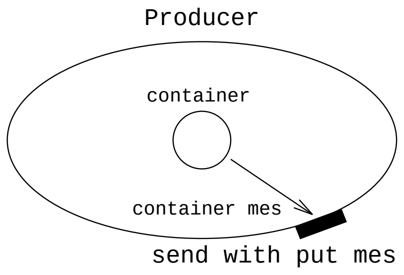


Figure 2.9: The `Producer` Object

For the sake of simplicity, we did not model the creation of messages in the `Producer` object specification (see Figure 2.9). Instead, we simply gave an initial value for the `container` place, which is a multi-set of three messages represented by the three free variables `m1`, `m2` and `m3`. Free variables may appear in the **Initial** field and represent any value of the mentionned sort. Thus, the `send` method asks to be synchronized with the `put` method of the `Buffer` object. The synchronization mechanism unifies both the "`put m`" and "`put mes`" expressions, and if the `put` event can occur the synchronization is then performed and a message is transmitted by the producer to the buffer. Figure 2.11 summarizes the synchronization requests which connect the three objects involved in a producer-consumer system. The arrows represent the dependency relationship, known also as client-ship relation, but not the data flow.

```
Object Consumer;
Interface
  Method receive;
Body
  Use Message, Buffer;
  Place container _ : message;
  Axioms
  receive With get m : -> container m;
  Where
  m : message;
End Consumer;
```

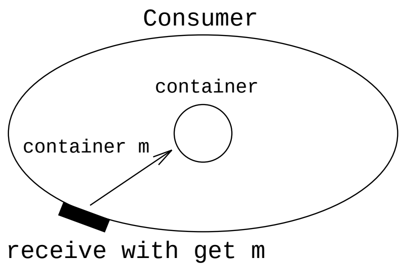


Figure 2.10: The `Consumer` Object

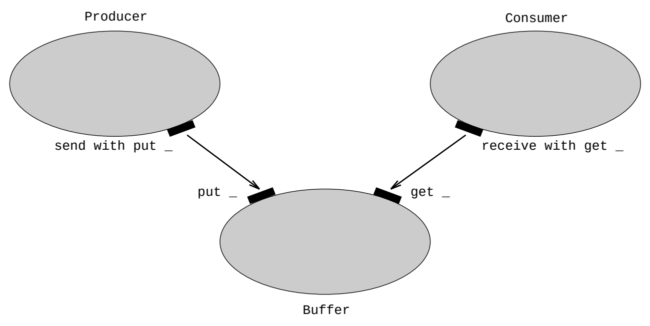


Figure 2.11: An Outline of the Producer-Consumer System

As with algebraic specifications, genericity over objects is allowed but, nevertheless, the formal parameters can be either abstract data type modules or object modules. Here we only give a minor example of a generic object which depends upon an ADT module. In Figure 2.8 we gave the specification of the buffer object, based upon the specification of a fifo of messages. We now propose the specification of a similar, but simpler, object using genericity. The considered generic object is a variant of the buffer, in which the fifo data structure is simply removed. We call this generic object `Bag`, as is shown in Figure 2.12, because it is based exclusively upon the implicit multi-set structures of the places. The multi-set data structure induces some different concurrent behaviors, as will be seen in the following examples. Object `BagMessage` represents the instantiation of the `Bag` generic object by means of the module `Message`.

The following examples illustrate some transition/method firings and several synchronization schemes which are more complex than those previously presented in the producer-consumer example. Figure 2.13 depicts three objects which have some slight differences. The left object `Obj1` shows two independent methods in which synchronizations may occur independently, nay, simultaneously. `Obj2` refers to the spontaneous reaction of an object. Indeed, internal transitions behave differently from methods, in the sense that internal transitions, if they can be fired, are fired at any time. For example, if a synchronization is requested for method `m1` of the object `Obj2`, then a value will be put into the place `p1` and immediately transfered into the place `p2`. This phenomena will not appear in the object `Obj3`, because the internal transition `t` has been replaced by a method `m`. The same remark holds for Figure 2.14 in which the internal transition `t` of the object `Obj4` asks to be synchronized with a method `m`, while within the object `Obj5` the synchronization is requested by a method `m'`. In other words the synchronization request of `Obj4` is perform as soon as the

```
Generic Object Bag(Elem);
Interface
  Methods get _ , put _ : elem;
Body
  Place  bag _ : elem;
  Axioms
  put e : -> bag e;
  get e : bag e -> ;
  Where
  e : elem;
End Bag;
```

```
Object BagMessage As Bag(Message);
Morphism elem -> message;
Interface
Body
End BagMessage;
```

Figure 2.12: The Generic Version of the Object `Bag` and its Instantiation

transition `t` can be fired, while that of `Obj5` is dependent upon an external event.

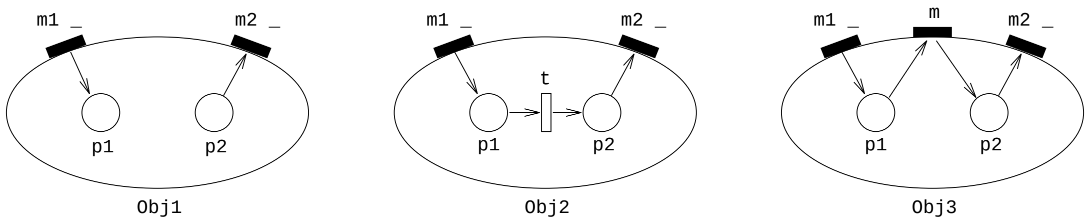


Figure 2.13: Method Firing Versus Transition Firing

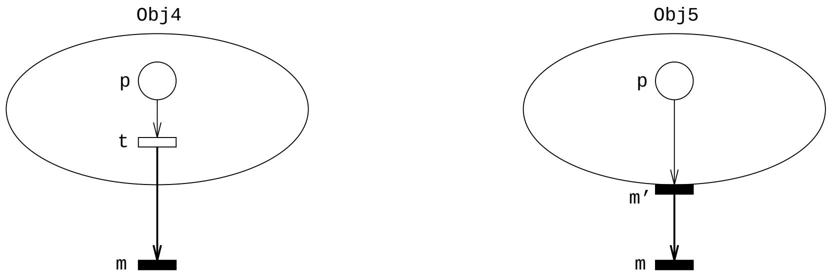


Figure 2.14: External and Internal Synchronizations

Figure 2.15 illustrates two simple synchronization requests. On the left, method `m1` of object `O1` asks to be simultaneously synchronized with methods `m2` and `m3` of both of the distinct objects, `O2` and `O3`, by means of the expression "`m1 with m2 // m3`". This expression must be read as follows: method `m1` behaves the same as the behavior of the simultaneous execution of `m2` and `m3` plus some local condition on `O1`. On the right, the synchronization involves the same object twice by means of the expression "`m1` **`With`**`m2 .. m3`", which corresponds to the execution of `m2` followed by the execution of `m3`.

Figure 2.16 depicts a synchronization request involving the three synchronizations operators which are provided by the formalism. Once again, the synchronization expression

`m1` **`With`**`m2 // ((m3 .. m4) + m5)`

must be read as follows: method `m1` behaves just as simultaneous methods `m2` and the alternative choice between `m3` followed by `m4` or `m5`.

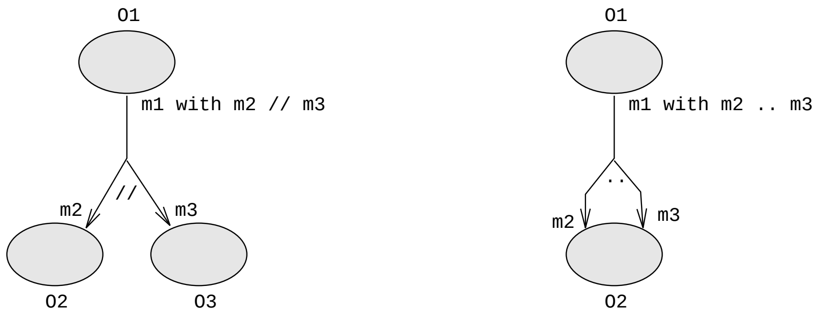


Figure 2.15: two Synchronization Schemes

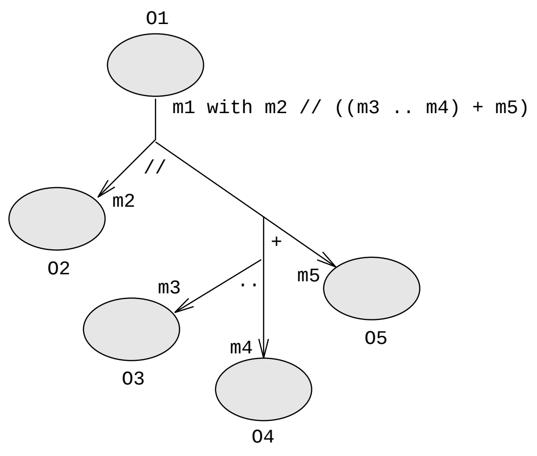


Figure 2.16: One More Involved Synchronization Scheme

3 Specification Examples

Giovanna Di Marzo & Olivier Biberstein

3.1 Introduction

This part presents some examples which illustrate the use of CO-OPN$_{1.5}$ specifications, and in particular the synchronization mechanism, the use of the **Interface**, **Body** parts, as well as the instantiations possibilities and the use of infix notation for the names.

The examples presented here consist of: (1) the Alternating Bit Protocol which enables two entities to exchange messages across unreliable channels; (2) the fair Lift, which processes calls from inside and/or outside a cabin without causing users to wait an infinite amount of time for the lift; (3) the Hierarchical Routing which is based here upon a two-level routing hierarchy.

Relevant CO-OPN$_{1.5}$ specifications for these examples are collected in Section 3.5.

3.2 Alternating Bit Protocol

The *Alternating Bit Protocol* (ABP) allows two communicating entities to exchange messages alternatively across an unreliable transmission medium. In this protocol, messages can be lost or altered. The existence of an altered message detection mechanism is assumed and a classic time-out retransmission mechanism is used for coping with the problem of lost messages. When an entity receives a corrupted message, it is rejected, thus reducing 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 only one receiver, who in turn acknowledges the messages it receives correctly. We also assume that the transmission lines consist of two unreliable FIFO queues.

Each time the sender initiates the transmission of a message, it activates a timer, which is not modeled in CO-OPN$_{1.5}$ but assumed external. If no acknowledgment is received within the given time limit 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 is set too short.

In order to circumvent the problem of duplication, messages are numbered sequentially with modulo-2 sequences. Consequently, each message, apart from containing error detection information, also contains 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 that it is receiving a duplicated message. In order to allow the sender and the receiver to synchronize, the receiver also returns acknowledgment for duplicated messages. This 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.

Figure 3.1 depicts two entities, a Sender of messages and a Receiver of messages, which are connected across two unreliable channels.

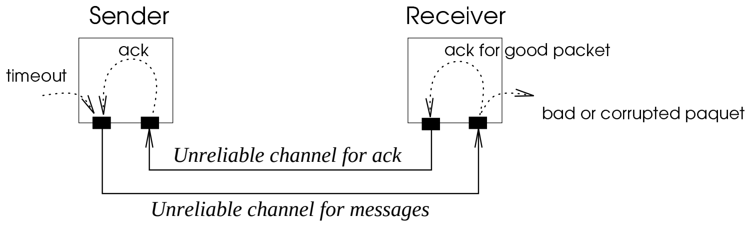


Figure 3.1: Alternating Bit Protocol

CO-OPN$_{1.5}$ Modeling

Messages contain no data and it is only the sort `message` which composes messages as is shown in Figure 3.4. In the same figure we observe the modules `Altbit` and `Frame`, which correspond, respectively, to the message number (modulo-2) and the transmitted frame (a message coupled with its number).

The final abstract data type in the ABP example is a first-in-first-out buffer of frame used for modeling communication links as is shown in Figure 3.5.

The first entities in our example which have an internal state are both the links `EtherIn` and `EtherOut` objects. Figures 3.6 and 3.7 show the specifications of these objects. They represent the channel for messages and the channel for acknowledgments respectively.

The Sender entity is represented by the `Transmitter` object shown in Figure 3.9, while the Receiver entity is given by the `Receiver` object shown in Figure 3.8. They synchronize their emission, or reception, of messages across the two channels.

3.3 Lift

We present here an example of a lift moving between floors of a building. It is propelled first by calls, made either from the inside or the outside of the cabin from each floors, and then, secondly, by its direction. Indeed, the lift follows an up or down direction, so that during its motion it can provide for users who wish to take the lift in the same direction than those of the lift.

Figure 3.2 shows a lift in a building of 5 floors. Calls made outside the cabin indicate that a user wishes to go up or down, while calls made inside the cabin only indicate the floor to which the user wishes to go, because the lift can rest at any floor when the user performs its call from inside the cabin.

CO-OPN$_{1.5}$ Modeling

The `Direction` abstract data types, of figure 3.10, used by the Lift specification is made of 3 constants which represent three possible movement directions: up, down and an undefined direction. The `Floors` abstract data type of figure 3.12 states the floors by name: each floor is given a reference with a constant `Ni`, which indicates the i-th floor. The two fundamental abstract data types are `Goal` and `ListGoals` given by figure 3.11 and 3.13 respectively. The `Goal` type specifies cartesian products of directions and floors. Goals represent the wishes of users the lift has to satisfy.

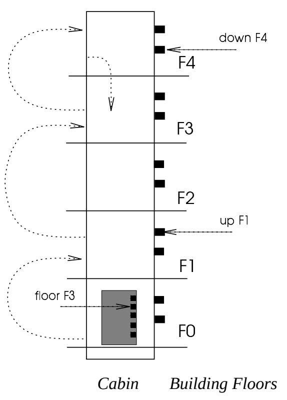


Figure 3.2: Lift

The `ListGoals` type specifies the lists of these goals. The `ListGoals` type encodes the way in which the lift (1) stores the goals before processing them, (2) processes its goals, (3) modifies its direction and position. The lift acts in a fair way, i.e. it follows its direction (up or down) until all goals containing this direction have been processed, then either there are no more goals and the lift stops and its direction becomes undefined, or it goes in the opposite direction to process the remaining goals.

Three objects model the lift. First of all the `Cabin` in Figure 3.14, which is made of buttons `CabinButtons` (one for each floor) and which stores the calls occurring inside the cabin in its memory, called `MemCabin`. These calls are stored as goals, composed of both the undefined direction and the floor chosen by the user. Once calls have been stored, the cabin forwards them to the `Control` object of Figure 3.16, which stores them in the list of goals of the lift and will process them later. The second object is the `BuildingFloors` of Figure 3.15 which acts as the cabin, with the difference that it is dedicated for processing the calls external to the cabin. The difference resides in the fact, that outside the cabin, a user must mention if it wants to go up or down. Finally, the third object `Control` processes calls that have been forwarded by either the `Cabin` or `BuildingFloors`. According to the list of goals the cabin must perform and to the current position and direction of the lift, the `Control` object is able to update the position, direction and goals of the lift.

3.4 Hierarchical Routing

Hierarchical routing is necessary in large networks, in which the gateways cannot maintain huge routing tables. Hosts are divided into regions, and the packets are routed first of all between regions till they reach their destination region, and then routed onwards to the destination host inside the destination region. Based upon the size of the whole network a two- (or more) level hierarchy of routes is necessary [Tan89].

Our model involves three regions, Region_0, Region_1, Region_2 working with a two-level hierarchy of routes. Region_1 and Region_2 contain subregions: Region_{1i}, Region_{2i} for $i = 1, 2$. The regions are organized as a binary tree, where each node act as a default gateway for its two children (sub-)regions. The default gateway is responsible for (1) routing each packet, destination address of which concerns its children region, for (2) routing all unknown packets to its default gateway (its parent region), and for (3) consuming each packet addressed to its region.

Figure 3.3 shows the above configuration. Packets going from Region_{12} and having to reach

Region$_{22}$ will first of all be routed towards the default gateway of Region$_{12}$, which is Region$_1$. Similarly they will be routed to the next default gateway, Region$_0$, which identifies them as being addressed to the sub-tree of Region$_2$. From Region$_2$ they will finally be routed to their destination in Region$_{22}$.

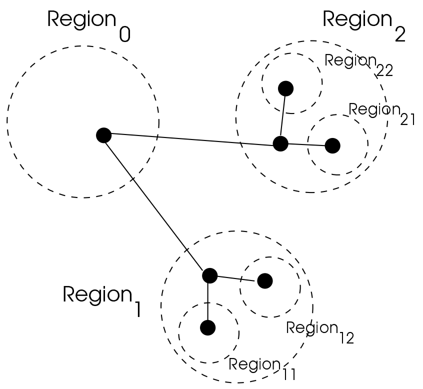


Figure 3.3: Hierarchical Routing

We have chosen to model only the default gateways and their work to route packets among the regions. Of course, each region is made of several users and the default gateway, instead of only consuming messages addressed to its region should also distribute them among all the users.

CO-OPN$_{1.5}$ Modeling

The abstract data type allowing the above modeling of hierarchical routing is a hierarchical type of addresses, represented by the `Address` abstract data type of Figure 3.17. In a n-level of routing, addresses are made of n parts $a_1 - a_2 - \ldots - a_n$, where a_1 means the *super* region in which the recipient resides, a_2 indicates which subregion inside region a_1 is referenced, and so on until a_n which indicates what user is involved in the Region a_{n-1}. Our example shows a two-level of routing, so that addresses used in the specifications have only one or two parts.

The generic object `User` in Figure 3.18 models the behavior of a gateway, which is able to route packets to its children or to its default gateway. A `User` object communicates with its default gateway (another `User` object) through methods `SendToGW` and `GetFromGW`. It communicates with its 2 children using transitions `GetFromUseri`, `SendToUseri`, $i = 1, 2$, and consumes messages for itself with `Consume`.

Three instantiations `Useri`, $i = 0, 1, 2$, stand for the three interconnected regions, with `User0` being the default gateway between the other two regions. Figures 3.19, 3.20 represent `User0` and `User1`.

Axioms of these objects specify the routing: given their address, messages will take the known route or will be forwarded to the default gateway.

Four objects `Userij`, $i = 1, 2$, $j = 1, 2$, lie at the bottom of the hierarchy. They are the leaves of the binary tree and for this reason they don't have any children to whom messages have to be forwarded. They are only consumers of messages and forwarders of unknown messages to their parent. Figures 3.21 and 3.22 represent objects `User11` and `User12`, which are the children subregions of `User1`.

3.5 CO-OPN$_{1.5}$ Specifications

3.5.1 Specification of the Alternate Bit Protocol Example

```
Adt Message;
Interface
  Sort message;
Body
End Message;
```

```
Adt Frame;
Interface
  Use Message, Altbit;
  Sort frame;
  Generator
    comp _ _ : message altbit -> frame;
Body
End Frame;
```

```
Adt Altbit;
Interface
  Use Booleans;
  Sort altbit;
  Generators
    0 : -> altbit;
    1 : -> altbit;
  Operation
    inc _ : altbit -> altbit;
Body
  Axioms
    inc 0 = 1;
    inc 1 = 0;
End Altbit;
```

Figure 3.4: Three Basic Abstract Data Types of the ABP Example

```
Adt Fifo;
Interface
  Use  Frame, Booleans, Altbit;
  Sort fifo;
  Generators
    empty      : -> fifo;
    put _ _    : fifo frame -> fifo;
  Operations
    empty?   _ : fifo -> boolean;
    rm  _      : fifo -> fifo;
    get _      : fifo -> frame;
    numframe _ : fifo -> altbit;
Body
  Axioms
    empty? empty = true;
    empty? (put ff fr) = false;

    rm empty = empty;
    rm (put empty fr) = empty;
    empty? ff = false => rm (put ff fr) = put (rm ff) fr;

    get (put empty fr) = fr;
    empty? ff = false => get (put ff fr) = get ff;
    empty? ff = false => numframe (put ff fr) = numframe ff;

    numframe(put empty (comp msg num)) = num;
  Where
    ff  : fifo;
    fr  : frame;
    msg : message;
    num : altbit;
End Fifo;
```

Figure 3.5: `Fifo` of Frames Abstract Data Type

```
Object EtherIn;
Interface
  Use Message, Altbit;
  Methods
    PutFrame _ _ ,
    GetFrame _ _ : message altbit;
    LostFrame;
Body
  Use Frame, Fifo;
  Place
    OnChannelFrame _ : fifo;
  Initial
    OnChannelFrame empty;
  Axioms
    PutFrame msg frnum :
     OnChannelFrame ffch -> OnChannelFrame (put ffch (comp msg frnum));

    GetFrame msg frnum :
      OnChannelFrame (put ffch (comp msg frnum))
      -> OnChannelFrame (rm (put ffch (comp msg frnum)));

    LostFrame :
      OnChannelFrame ffch -> OnChannelFrame (rm ffch);
  Where
    ffch  : fifo;
    frnum : altbit;
    msg   : message;
End EtherIn;
```

Figure 3.6: The `EtherIn` Object

```
Object EtherOut;
Interface
  Use Message, Altbit;
  Methods
    GetAck _ _ ,
    PutAck _ _ : message altbit;
    LostAck;
Body
  Use Frame, Fifo;
  Place
    OnChannelAck _ : fifo;
  Initial
    OnChannelAck empty;
  Axioms
    GetAck msg frnum :
      OnChannelAck (put (ffch (comp msg frnum)))
      -> OnChannelAck (rm (put (ffch (comp msg frnum))));

    LostAck:
      OnChannelAck ffch -> OnChannelAck (rm ffch);

    PutAck msg frnum :
      OnChannelAck ffch -> OnChannelAck (put ffch (comp msg frnum));
  Where
    fr    : frame;
    ffch  : fifo;
    frnum : altbit;
    msg   : message;
End EtherOut;
```

Figure 3.7: The `EtherOut` Object

```
Object Receiver;
Interface
  Use Altbit;
  Methods
    TxAck      _ ,
    OkFrameRcv _ : altbit;
Body
  Use Message, Frame, Fifo, EtherIn, EtherOut;
  Place
    FrameRcv _ : fifo;
  Initial
    FrameRcv empty;
  Axioms
    TxAck b With PutAck msg b :
      FrameRcv (put ffrcv (comp msg b)) -> FrameRcv ffrcv;

    OkFrameRcv b With GetFrame msg b :
      FrameRcv ffrcv -> FrameRcv (put ffrcv (comp msg b));
  Where
    msg   : message;
    ffrcv : fifo;
    b     : altbit;
End Receiver;
```

Figure 3.8: The Receiver Object

```
Object Transmitter;
Interface
  Use Altbit;
  Methods
    DupAckRcv _ ,
    TxFrame   _ ,
    RetxFrame _ : altbit;
    ProcAck;
    OkAckRcv;
Body
  Use Message, Frame, Fifo, EtherIn, EtherOut;
  Places
    Idle    _ ,
    TimeOut _ ,
    AckRcv  _ ,
    Last    _ : altbit;
  Initial
    Idle 0, Last 1;
  Axioms
    acknum = inc frnum => DupAckRcv frnum With GetAck msg frnum :
      Last acknum -> Last frnum;

    ProcAck :
      AckRcv acknum, TimeOut frnum -> Idle acknum;

    OkAckRcv With GetAck msg frnum :
      Last frnum -> AckRcv frnum, Last (inc frnum);

    TxFrame frnum With PutFrame msg (inc frnum) :
      Idle frnum -> TimeOut (inc frnum);

    RetxFrame frnum With PutFrame msg frnum :
      TimeOut frnum -> TimeOut frnum;
  Where
    msg    : message;
    acknum : altbit;
    frnum  : altbit;
End Transmitter;
```

Figure 3.9: The `Transmitter` Object

3.5.2 Specification of the Lift Example

```
Adt Direction;
Interface
  Use  Booleans;
  Sort direction;
  Generators
    UpDirection         ,
    DownDirection       ,
    UndefinedDirection : -> direction;
  Operation
    _ = _ : direction direction -> boolean;
Body
  Axioms
    UpDirection = UpDirection               = true;
    UpDirection = DownDirection             = false;
    UpDirection = UndefinedDirection        = true;

    DownDirection = UpDirection             = false;
    DownDirection = DownDirection           = true;
    DownDirection = UndefinedDirection      = true;

    UndefinedDirection = DownDirection      = true;
    UndefinedDirection = UpDirection        = true;
    UndefinedDirection = UndefinedDirection = true;
End Direction;
```

Figure 3.10: The `Direction` ADT of the Lift Object

```
Adt Goal As Pc2(Direction, Floors);
Morphism
  _ = _ in ComparableItem2 -> _ = _ in Floors;
Rename
  pc2     -> goal;
  fst _   -> whichDirection _ ;
  snd _   -> whichFloor _ ;
  < _ _ > -> pair _ _ ;
Interface
Body
End Goal;
```

Figure 3.11: The `Goal` ADT of the Lift Object

```
Adt Floors;
Interface
  Use  Booleans;
  Sort floors;
  Generators
    F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10,
    notdefined : -> floors;
  Operations
    last,  first         : -> floors;
    next _ ,  previous _ : floors -> floors;
    _ notdefined?        : floors -> boolean;
    _ = _ ,
    _ < _ : floors floors -> boolean;
Body
  Axioms
    first = F0;            last  = F10;

    ;; next, if x >= last then next x = notdefined
    ;; previous, if x <= first then previous x = notdefined
    next F0  = F1;          previous F1  = F0;
    next F1  = F2;          previous F2  = F1;
    next F2  = F3;          previous F3  = F2;
    next F3  = F4;          previous F4  = F3;
    next F4  = F5;          previous F5  = F4;
    next F5  = F6;          previous F6  = F5;
    next F6  = F7;          previous F7  = F6;
    next F7  = F8;          previous F8  = F7;
    next F8  = F9;          previous F9  = F8;
    next F9  = F10;         previous F10 = F9;
    next F10 = notdefined;  previous F0  = notdefined;

    next notdefined = notdefined;
    previous notdefined = notdefined;

    F0  notdefined? = false;    F1  notdefined? = false;
    F2  notdefined? = false;    F3  notdefined? = false;
    F4  notdefined? = false;    F5  notdefined? = false;
    F6  notdefined? = false;    F7  notdefined? = false;
    F8  notdefined? = false;    F9  notdefined? = false;
    F10 notdefined? = false;    notdefined notdefined? = true;

    x notdefined? and y notdefined?           = true => x = y = true;

    (not x notdefined?) and y notdefined?     = true => x = y = false;

    x notdefined? and not y notdefined?       = true => x = y = false;

    (not x notdefined?) and not y notdefined? = true =>
      (x = y) = (previous x = previous y);

    x notdefined? and y notdefined?           = true => x < y = false;

    (not x notdefined?) and y notdefined?     = true => x < y = false;

    x notdefined? and not y notdefined?       = true => x < y = true;

    (not x notdefined?) and not y notdefined? = true =>
      x < y = previous x < previous y;
  Where
    x, y : floors;
End Floors;
```

Figure 3.12: The `Floors` ADT of the Lift Object

```
Adt ListGoals As List(Goal);
Rename
  []    -> emptylist;
  _ ' _ -> cons _ _ ;
  _ | _ -> _ cat _  ;
  # _   -> length _ ;
  list  -> listGoals;
Interface
  Use Booleans, Floors, Direction;
  Operations
    insertGoal _ _        : listGoals goal             -> listGoals;
    reachGoal  _ _ _      : listGoals floors direction -> listGoals;
    move _ _ _            : listGoals floors direction -> floors;
    modifyDirection _ _ _ : listGoals floors direction -> direction;
    existLT _ _ ,
    existGT _ _ ,
    existEqual _ _        : listGoals floors -> boolean;
    existGoal  _ _        : listGoals goal   -> boolean;
Body
  Axioms
    insertGoal emptylist gl = cons gl emptylist;

    whichFloor gl < whichFloor val = true  =>
        insertGoal (cons val lst) gl  = cons gl (cons val lst);

    whichFloor gl < whichFloor val = false =>
        insertGoal (cons val lst) gl  = cons val (insertGoal lst gl);

    ;; no goals for reachGoal
    reachGoal emptylist fl d = emptylist;

    ;; the floor and the direction are a correct goal
    (whichFloor gl = fl and whichDirection gl = d) = true =>
        reachGoal (cons gl lst) fl d = lst;

    ;; the floor is reached but the direction is wrong,
    ;; the list of goals is explored
    (whichFloor gl = fl and not whichDirection gl = d) = true =>
        reachGoal (cons gl lst) fl d = cons gl (reachGoal lst fl d);

    ;; the floor is not reached , the list of goals is explored
    whichFloor gl = fl = false =>
        reachGoal (cons gl lst) fl d = cons gl (reachGoal lst fl d);

    move emptylist fl d = fl;

    d = UndefinedDirection => move (lst, fl d) = fl;

    ;; the current floor is a goal, there is not any other one
    (existEqual lst fl and length lst = succ 0) = true =>
        move (lst, fl d) = fl;

    ;; the current floor is a goal, verification of the direction
    (existEqual lst fl and existGoal lst (pair d fl)) = true =>
        move (lst, fl d) = fl;
```

```
;; the current floor is a goal,  incorrect direction (up direction)
(existEqual lst fl and not existGoal lst (pair d fl)) and
  existGT lst fl = true  =>
    move (lst, fl UpDirection) = next fl;

(existEqual lst fl and not existGoal lst (pair d fl)) and
  not existGT lst fl = true =>
    move (lst, fl UpDirection) = fl;

;; the current floor is a goal,  incorrect direction (down direction)
(existEqual lst fl and not existGoal lst (pair d fl)) and
  existLT lst fl = true =>
    move (lst, fl DownDirection) = previous fl;

(existEqual lst fl and not existGoal lst (pair d fl)) and
  not existLT lst fl = true =>
    move (lst, fl DownDirection) = fl;

;; the current floor is not a goal
existEqual lst fl = false => move (lst, fl UpDirection) = next fl;

existEqual lst fl = false => move (lst, fl DownDirection) = previous fl;

;; the current floor is not a goal
;; modifyDirection depends on the direction
existGT lst fl = true =>
    modifyDirection (lst, fl UpDirection) = UpDirection;

(existLT lst fl) and not existGT lst fl = true =>
    modifyDirection (lst, fl UpDirection) = DownDirection;

(not existLT lst fl) and not existGT lst fl = true =>
    modifyDirection (lst, fl UpDirection) = UndefinedDirection;

existLT lst fl = true =>
    modifyDirection (lst, fl DownDirection) = DownDirection;

(not existLT lst fl) and existGT lst fl = true =>
    modifyDirection (lst, fl DownDirection) = UpDirection;

(not (existLT lst fl)) and not (existGT lst fl) = true =>
    modifyDirection (lst, fl DownDirection) = UndefinedDirection;

existGT lst fl = true  =>
    modifyDirection (lst, fl UndefinedDirection) = UpDirection;

existLT lst fl and not existGT lst fl = true =>
    modifyDirection (lst, fl UndefinedDirection) = DownDirection;

(not existLT lst fl) and not existGT lst fl = true =>
    modifyDirection (lst, fl UndefinedDirection) = UndefinedDirection;

;; Does it exist a lower floor
existLT emptylist fl  = false;

whichFloor gl < fl = true  => existLT (cons gl lst) fl = true;

whichFloor gl < fl = false => existLT (cons gl lst) fl = existLT lst fl;
```

```
    ;; Does it exist a floor
    existEqual emptylist fl = false;

    whichFloor gl = fl = false => existEqual (cons gl lst) fl = existEqual lst fl;

    whichFloor gl = fl = true => existEqual (cons gl lst) fl = true;

    ;; Does it exist a greater floor
    existGT emptylist fl = false;

    whichFloor gl < next fl = false => existGT (cons gl lst) fl = true;

    whichFloor gl < next fl = true  => existGT (cons gl lst) fl = existGT lst fl;

    ;; Does it exist a goal
    existGoal emptylist gl = false;

    (head lst = gl) = true  => existGoal lst gl = true;

    (head lst = gl) = false => existGoal lst gl = existGoal (tail lst) gl;
  Where
    gl, val : goal;
    lst     : listGoals;
    fl      : floors;
    d       : direction;
End ListGoals;
```

Figure 3.13: The `ListGoals` ADT of the Lift Object

```
Object Cabin;
Interface
  Use Control;                        ;; Synchro from (performcall )
  Methods
    floor _ : floors ;
Body
  Use
    Direction, Floors, Booleans;
    Goal;                               ;; list of users
  Transition
    performfloor;                       ;; !! synchro
  Places
    CabinButtons _ : floors;
    MemCabin _     : goal;
  Initial
    ;; Buttons inside the cabin
    CabinButtons F0, CabinButtons F1, CabinButtons F2, CabinButtons F3, CabinButtons F4;
  Axioms
    ;; Store a call occured inside the cabin into the memory of the
    ;; cabin. The direction is undefined as the cabin can currently
    ;; lie at any floor.
    floor bfl:
      CabinButtons bfl
      -> CabinButtons bfl, MemCabin (pair UndefinedDirection bfl);

    ;; Forward a call occured inside to the cabin to the Control object
    performfloor With call d bfl:
      MemCabin (pair d bfl)
      -> ;
  Where
    bfl : floors;
    d   : direction;
End Cabin;
```

Figure 3.14: The `Cabin` Object of Lift

```
Object BuildingFloors;
Interface
  Use Control;                        ;; Synchro from (performcall)
  Methods
    down _ : floors ;
    up   _ : floors ;
Body
  Use
    Direction, Floors, Booleans;
    Goal;                               ;; list of users
  Transition
    performcall;                        ;; !! synchro
  Places
    FloorButtons _ : floors;
    MemFloors    _ : goal;
  Initial
    ;; Buttons to call cabin from the floors
    FloorButtons F0, FloorButtons F1, FloorButtons F2, FloorButtons F3, FloorButtons F4;
  Axioms
    ;; Store in the memory of the Floors at which floor a
    ;; user has called the cabin b and for which direction up down
    down b :
      FloorButtons b
      -> FloorButtons b, MemFloors (pair DownDirection b);

    up b :
      FloorButtons b
      -> FloorButtons b, MemFloors (pair UpDirection b);

    ;; Forward the call to the Control object
    performcall With call d b :
      MemFloors (pair d b)
      -> ;
  Where
    b : floors;
    d : direction;
End BuildingFloors;
```

Figure 3.15: The BuildingFloors Object of Lift

```
Object Control;
Interface
  Use Direction, Floors;
  Methods
    call _ _ : direction, floors ;
Body
  Use
    Booleans;
    Goal, ListGoals;            ;; list of users
  Transitions
    moving;
  Places
    GoalsToSolve _ : listGoals;
    Moving _       : direction;
    Position _     : floors;
  Initial
    GoalsToSolve emptylist, Moving UndefinedDirection, Position F0;
  Axioms
    ;; Move the position of the cabin if the cabin is wanted to
    ;; go somewhere
    (empty? l = false) = true => moving:
       Position fl, Moving dir, GoalsToSolve l
       -> Moving (modifyDirection l fl dir),
          GoalsToSolve (reachGoal l fl dir),
          Position (move l fl dir) ;

    ;; Store forwarded call from Cabin or BuildingFloors
    call dir fl :
      GoalsToSolve l
      -> GoalsToSolve (insertGoal (l (pair dir fl))) ;
  Where
    fl  : floors;
    dir : direction;
    l   : listGoals;
End Control;
```

Figure 3.16: The `Control` Object of Lift

3.5.3 Specification of the Hierarchical Routing

```
Adt Address;
Interface
  Use  Booleans;
  Sort address;
  Generators
    0, 1, 2, notdefined :    -> address;
    _ - _  : address address -> address;
  Operation
    _ = _  : address address -> boolean;
    head _ ,
    tail _ : address -> address;
Body
  Axioms
    0 = 0 = true;    1 = 1 = true;    2 = 2 = true;

    notdefined = notdefined = true;

    0 = 1 = false;   0 = 2 = false;   0 = notdefined = false;
    1 = 0 = false;   1 = 2 = false;   1 = notdefined = false;
    2 = 0 = false;   2 = 1 = false;   2 = notdefined = false;

    notdefined = 0 = false;
    notdefined = 1 = false;
    notdefined = 2 = false;

    x-y = 0=false;    x-y = 1=false;    x-y = 2=false;
    0 = x-y=false;    1 = x-y=false;    2 = x-y=false;
    ((x-y)=(x1-y1)) = (x=x1) and (y=y1);

    head 0=0;
    head 1=1;
    head 2=2;
    head notdefined=notdefined;
    head (0-x)=0;
    head (1-x)=1;
    head (2-x)=2;

    tail 0=notdefined;
    tail 1=notdefined;
    tail 2=notdefined;
    tail notdefined = notdefined;
    tail (0-x)=x;
    tail (1-x)=x;
    tail (2-x)=x;
  Where
    x, y, x1, y1 : address;
End Address;
```

Figure 3.17: The `Address` ADT for Hierarchical Routing

```
Generic Object User;
Interface
  Use Address;
  Methods
    SendToGW  _ : address;
    GetFromGW _ : address;
Body
  Transitions
    SendToUser1 ;
    GetFromUser1;
    SendToUser2 ;
    GetFromUser2;
    Consume     ;
  Places
    User : address;
End User;
```

Figure 3.18: Generic User of Hierarchical Routing

```
Object User0 As User;
Interface
Body
  Use User1, User2;
  Initial
    User 0; User 1-2; User 1-1; User 2-1;
  Axioms
    ;; Forward messages to its User1 or User2
    head adr = 1 = true => SendToUser1 With GetFromGW In User1 adr : User adr -> ;

    head adr = 2 = true => SendToUser2 With GetFromGW In User2 adr : User adr -> ;

    ;; Message is for current User
    adr=0 = true => Consume: User adr -> ;

    ;; Collect messages for current User or messages to forward
    (not head adr = 1) = true =>
      GetFromUser1 With SendToGW In User1 adr : -> User adr;

    (not head adr = 2) = true =>
      GetFromUser2 With SendToGW In User2 adr : -> User adr;
  Where
    x, adr : address;
End User0;
```

Figure 3.19: An Instantiation of User: `User0` the Root User

```
Object User1 As User;
Interface
Body
  Use User11, User12;
  Initial
    User 0; User 2-1; User 2-2; User 2; User 1-1;
  Axioms
    ;; Forward messages to its User1 or User2
    head x = 1 = true => SendToUser1 With GetFromGW In User11 1-x : User 1-x -> ;

    head x = 2 = true => SendToUser2 With GetFromGW In User12 1-x: User 1-x -> ;

    ;; Message is for current User
    adr =1 = true => Consume : User adr -> ;

    ;; Collect messages for current User or messages to forward
    (not head adr = 1) or (not head tail adr = 1) = true =>
      GetFromUser1 With SendToGW In User11 adr : -> User adr;

    (not head adr = 1) or (not head tail adr = 2) = true =>
      GetFromUser2 With SendToGW In User12 adr : -> User adr;

    ;; Forward (Receive) messages to (from) gateway
    SendToGW  adr : User adr -> ;

    GetFromGW adr : -> User adr;
  Where
    x, adr : address;
End User1;
```

Figure 3.20: The Instantiation of User: `User1` an Intermediary Node

```
Object User11 As User;
Interface
Body
  Initial
    User 0, User 2-2, User 2, User 1, User 1-1, User 1-(1-2);
  Axioms
    ;; No Forward of messages to its User1 or User2
    ;; Message is for current User;
    adr=1-1 = true => Consume: User adr -> ;

    ;; No collect of messages from its User1 or User2
    ;; Forward (Receive) messages to (from) gateway
    SendToGW  adr : User adr -> ;

    GetFromGW adr : -> User adr;
  Where
    x, adr : address;
End User11;
```

Figure 3.21: A Leaf User

```
Object User12 As User;
Interface
Body
  Initial
    User 2, User 1, User 1-1, User 2-2;
  Axioms
    ;; No Forward of messages to its User1 or User2
    ;; Message is for current User;
    adr=1-2 = true => Consume : User adr -> ;

    ;; No collect of messages from its User1 or User2
    ;; Forward (Receive) messages to (from) gateway
    SendToGW  adr : User adr -> ;

    GetFromGW adr : -> User adr;
  Where
    x, adr : address;
End User12;
```

Figure 3.22: A Second Leaf User

4 The Editor

JACQUES FLUMET

4.1 Introduction

The graphic editor offers an environment for supporting the development of the CO-OPN$_{1.5}$ formalism. This tool can read text files and generates their internal representation, or the other way around: transform the graphic objects/modules into a textual form. Every components of the SANDS$_{1.5}$ development system may be invoked from the graphic editor.

The editor we present here evolved out of the graphic tool described in [BFR93b] and is still being developed. The complete definition of the initial graphic formalism (CO-OPN version 1) can be found in [Flu95]. Both this graphic editor and its ancestor have been developed by means of a the meta-CASE GraphTalk 2.5 [1].

4.2 Functional Description

4.2.1 Generalities

- *Graphic Formalism*
  The graphic formalism of CO-OPN$_{1.5}$ is evolved out of CO-OPN (version 1). It includes definitions of graphic signatures for object modules and ADT modules. It can also describe synchronizations between objects and dependencies which exist between objects and modules. Several views represent a CO-OPN$_{1.5}$ application and, these views are described below.

- *Graphic Signature of Object*
  For an object, this representation shows its methods, transitions and profile (Figure 4.1). If its methods/transitions are defined in the body (respectively the interface), they are drawn on the bottom (respectively the top) of the object.

- *Graphic Signature of ADT Module*
  For an ADT module, this representation shows its methods/generators and profile (Figure 4.2). If its methods/generators are defined in the body (respectively the interface), they are drawn on the bottom (respectively the top) of the object.

- *Algebraic Petri Nets*
  Each CO-OPN$_{1.5}$ object is defined by an Algebraic Petri net (Figure 4.3). Places (displayed as circles) are typed by a sort. This sort is displayed under the place. Transitions and methods are displayed with rectangles, with white representing transitions and black representing

[1]GraphTalk is a commercial product of Rank Xerox.

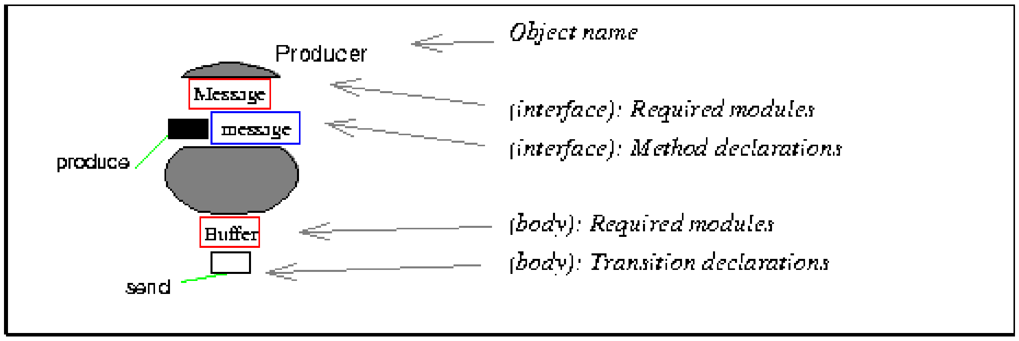

Producer
Object name
Message
message
(interface): Required modules
(interface): Method declarations
produce
Buffer
(body): Required modules
(body): Transition declarations
send

Figure 4.1: A Graphic Signature of an Object

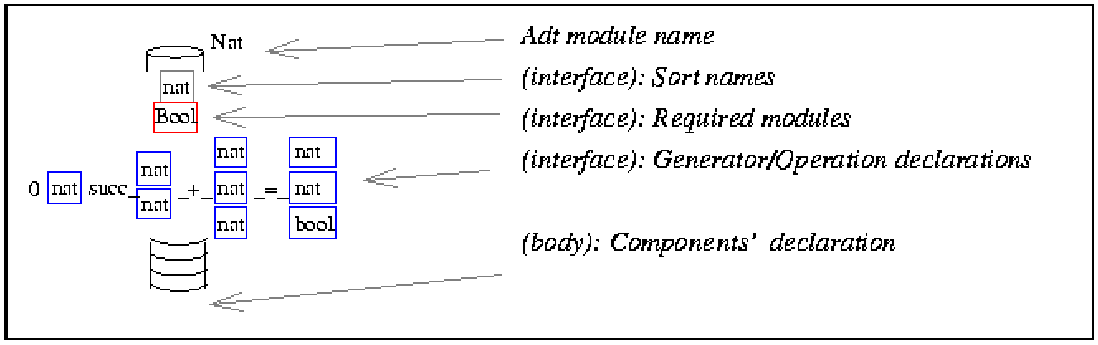

Nat
Adt module name
nat
(interface): Sort names
Bool
(interface): Required modules
0 nat succ_ nat _+_ nat _=_ nat bool
(interface): Generator/Operation declarations
(body): Components' declaration

Figure 4.2: A Graphic Signature of an ADT Module

methods. Edges are labeled with algebraic terms which represent tokens which are consumed or produced in the places.

- *Dependencies Graphs*
  This representation shows dependencies between objects and/or modules. Objects are displayed as ellipses, whereas modules are displayed as cylinders. An example is given in Figure 4.4.

4.2.2 Tool & Module Management

- *Data Consistency*
  The graphic editor maintains the consistency between names of modules, object and sort definitions. The dependency graphs are actually documentation graphs, which are snapshots of a CO-OPN$_{1.5}$ application described in the graphic editor.

- *Translation between Textual and Graphic Form (vice-versa)*
  The graphic editor can generate a program text corresponding to an object. It can also build a graphic representation from a textual form of an object or ADT module.

- *Interaction with other tools*
  The compiler and the simulator can be invoked directly from the graphic editor. If error messages appear, they are displayed in relation to the graphic component concerned.

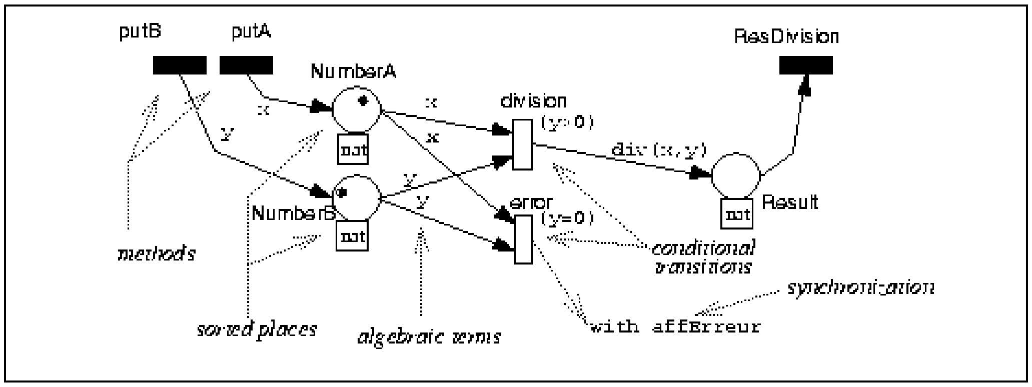


Figure 4.3: An Algebraic Petri Net (CO-OPN$_{1.5}$ object behavior description)

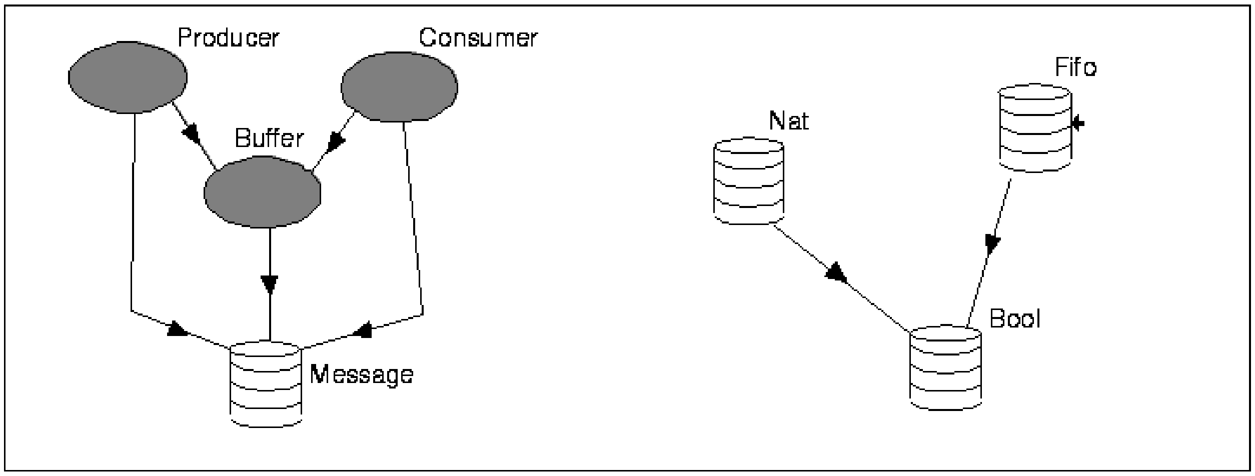


Figure 4.4: A Graph Dependency Relationship

4.3 Introduction Manual

4.3.1 Organization and Main Window

The main window displays all the kinds of views (left side of Figure 4.5) and the user can create instances of these views that generate new windows which are listed on the right side of the Figure 4.5.

- "`syst`" *Windows*
  This kind of windows allows the user to create ADT modules and objects. The "`Objects`" window is dedicated to the object creation and their associated methods as well as the method profiles. In a "`ModuleAlg`" window, the user creates ADT modules and their associated generators/operations, and can define their profiles.

- "`doc`" *Windows*
  A "`Dependences`" window is used for obtaining documentation on some modules or objects from the application. With a set of graphic operations, the user may mask or show particular details. A "`Marquage`" window can be used after a simulation session. The graphic editor is able to read a file generated by the simulator and to show the result in a graphic form. As mentioned for the "`Dependences`" window, the user may mask or show some particular details by using a set of graphic operations.

4.3.2 Working Session

Menus are attached to objects, ADT modules or windows in accordance with behavior characteristics. The graphic editor is designed for beginners who have no particular knowledge of the syntax of the CO-OPN$_{1.5}$ formalism. For advanced users, a syntactic textual editor can be used in collaboration with the graphic editor.

4.3.3 Actual Implementation

Up to date, a complete graphic editor as been developed for the SANDS (version 1) environment. A updated version exists for the SANDS$_{1.5}$, however some functionalities are missing. The development of a complete version is still in progress.

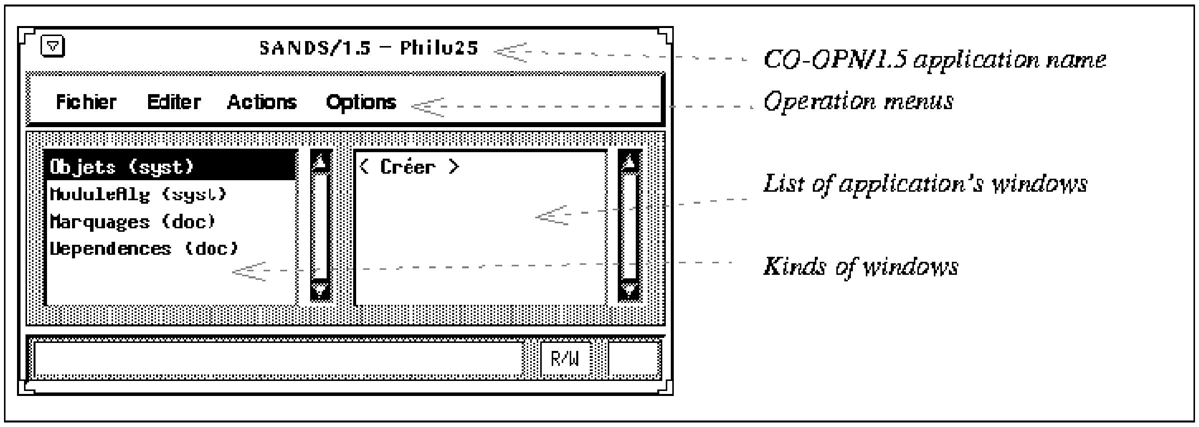


Figure 4.5: The Main Window of the Graphical Editor

5 The Checker

Didier Buchs

5.1 Introduction

The Checker is destined to check that a textual file contains a coherent part of a global specification, written in the CO-OPN$_{1.5}$ language. The principle used is mainly based on separated checking, which means that a module is checked in isolation, assuming that the dependency modules have been successfully checked. The smallest CO-OPN$_{1.5}$ tile specification that can be checked is the file which must contain at least one CO-OPN$_{1.5}$ module.

5.2 Functional Description

The Checker exists for checking that a textual file contains a coherent part of a global specification written in the CO-OPN$_{1.5}$ language. The syntax and the static semantics supported by the Checker are described in the annexes (syntax and static semantics).

The principle used is mainly based on separated checking and means that a module is checked in isolation, with the assumption that the dependency modules have been successfully checked. The smallest piece of CO-OPN$_{1.5}$ specification that can be checked is the file which must contain at least one CO-OPN$_{1.5}$ modules. CO-OPN$_{1.5}$ specifications are composed of four kinds of modules:

- ground or normal modules,
- parameter modules,
- instantiated modules,
- generic modules.

Each of these kinds of modules can be either algebraic or state based, that is to say described by structured algebraic nets.

The Checker tool performs two kinds of activities: the instantiation of the modules (partial or complete instantiation) and the checking of the entity definitions, as well as the checking of expressions of the specifications (functional axioms, behavioral axioms and theorems).

The instantiation process performs a translation of the instantiated module into a normal or instantiated module (for the partial instantiation), in which each reference to a parameter module identifier will be replaced by an identifier belonging to the concrete parameter module through the use of morphism. The use of renaming allows the changing of the resulting identifier of the module which is produced.

The checking process is divided into two steps. The first is destined to check that the definition of the involved entities such as the sorts, operations, generators, methods are well-formed and do not include any ambiguities. The second produces an abstract description of the expressions appearing into the different kind of axioms from the concrete syntax. The abstract description must be deterministic, so as to resolve possible conflicts as well as overloading.

The abstract description is used by the tools working downstream from the Checker, such as the Compiler, the Monitor and the Simulator.

5.3 Introduction Manual

The purpose of this part of the document is to explain how the checker can be used and the way in which it works. Example of checking are given, as well as examples of different kinds of errors. First of all, it is ensured that all environment variables are set correctly, according to the reference manual (variable SANDS).

The checker is easy to operate, since its use is reduced to a command line, where the possible options are described in the reference part below. The checker shows successively the modules that have been loaded, and the stages that have elapsed. Indication of possible errors is given by referring to the source file (line number) and axiom name.

5.3.1 Normal Use of the Ground Module

When files are correctly checked by the tools, a file is produced for each checked modules with the extension "`chck`" including a translation of the definitions and axioms into an abstract syntax in which possible ambiguities are removed. The abstract syntax is represented with a simple textual syntax in which each keyword represents the portion of the syntax which is being considered and the kind of identifier which is being declared. The abstract identifier is obtained by concatenation of the sub-identifier with the underline place holder. The prefix notation is used for the terms, instead of the mix-fix notation available in the CO-OPN$_{1.5}$ concrete syntax, thus reducing the cost of further term analysis.

Example:

```
Adt a;
Interface
  Sorts t,u;
  Operations
    f: t -> u;
Body
  Axioms
    ...;
  Where
    x : t;
End a;
```

```
Adt b;
Interface
  Sorts v,w;
  Operations
    f: v -> w;
Body
  Axioms
    ...;
  Where
    y : v;
End b;
```

Figure 5.1: Two Specifications

This example (Figure 5.1 and 5.2) produces the following abstract description shown in Figure 5.3.

Other modules, such as the parameter and generic modules are treated the same way, with the exception that the semantics rules are a little bit different.

```
Adt specuse;
Interface
Use a,b;
  Operations
    g: v -> w;
    _ inside? _ : u v -> t;
  Body
  Axioms
    ... =  f z;
  Where
    z : v;
End specuse;
```

Figure 5.2: The Use of the Previous Specifications

```
module_alg(data,'specuse').
interface_uses('specuse',['a','b']).
interface_operation('specuse','specuse.g:b.v -> b.w',['b.v'],'b.w').
interface_operation('specuse','specuse. _ isin _ :a.u,b.v -> a.t',
                    ['a.u','b.v'],'a.t').
...
body_axiom('specuse','specuse.ax0',[],...,'b.f:b.v -> b.w'('specuse.z')).
...
body_variable('specuse','specuse.z: -> b.v','b.v').
```

Figure 5.3: The internal form of the previous specifications

5.3.2 Normal Use of Instantiated Modules

When an instantiated module is given, the checking process requires two steps. First, instantiation is made using morphism and renaming, producing a set of new definitions and abstract terms. Then the checking process is applied again in order to find eventual errors coming from the faulty definitions of the morphism and renaming.

5.3.3 Possible Errors

Some errors can be found statically, according to the rules given in the annexes. The kind of possible errors concerns the incorrect syntactic definitions, the use of undefined identifier, the ambiguous or incoherent term typing (arity and sorting) and the re-definition of the identifier. Error messages are displayed, indicating which error has been encountered and where it has been found.

The error format has the following shape:

#errors#LINE[#AXIOM_NAME]#DESCRIPTION

The #LINE indicates the position of the error while [#AXIOM_NAME] indicates the axiom name considered, if appropriate. The #DESCRIPTION is a text describing the error with the description of the sub-portion of the line which produced the error.

6 The Simulator

Pascal Racloz & Didier Buchs

6.1 Introduction

The simulator is an important part of the SANDS$_{1.5}$ environment. Indeed, it allows the simulation of a CO-OPN$_{1.5}$ specification and the overall process is the following. From a global state of the system, composed of the union of the state of each object, it computes, under a specified set of constraints, its possible successor states. So, the simulation builds the reachability graph of the system made of those global states.

On the other hand, the algebraic data type part of the system's specification can also be simulated. Coarsely, an abstract data type can be evaluated.

This part describes the simulator tool for CO-OPN$_{1.5}$ specifications. It is structured essentially into three parts: the control environment settings, a section concentrated on visualization and simulation and the abstract data type behavior checker.

6.2 Functional Description

6.2.1 Generalities

- *Nets that can be Simulated*
  The nets that can be simulated are the usual Petri nets, algebraic nets (which include colored Petri nets) and hierarchical algebraic nets, i.e. CO-OPN$_{1.5}$ nets. It is also possible to produce, from a high level net, its underlying Petri net structure by taking the projection of its structured tokens on usual black tokens, and then to simulate them.

- *State and Transition*
  The state of a system is given by the set of the markings of the objects upon which the system is built. A transition from one state to another is labeled by the name of an object's method or by a synchronization expression.

- *Generic Markings*
  Due to the use of universally quantified variables within the terms of place contents, a marking can be viewed as a generic marking. Each instantiation of a free variable generates a particular marking. Figure 6.1 illustrate this notion. From the set of states represented by the generic marking M_2, only those represented by the generic marking SM_2 allow the firing of the given transition (arrow shown on the picture). This firing leads to the set of states represented by M_3.

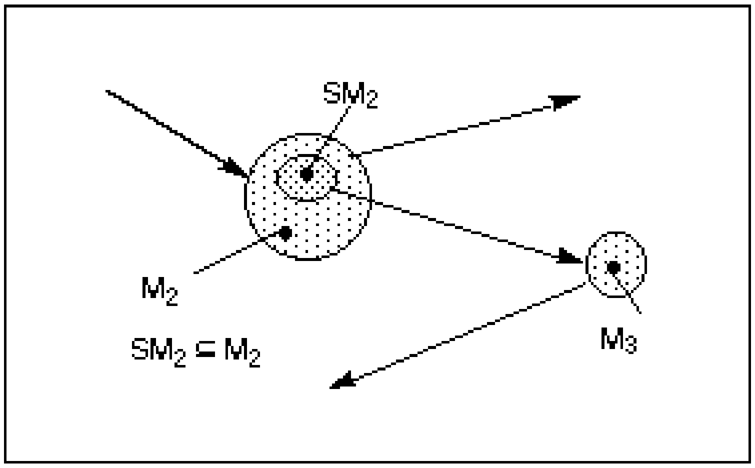


Figure 6.1: Generic Markings

- *Visibility*
  The simulation tool includes the notion of visibility. The user can specify the objects and for each object, the places and transitions that he would like to concentrate on and can hide the other items which are of no interest to him. 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.

- *Graphic Features*
  The simulator allows users to perform model specifications by using the classic features of a window manager (windows, buttons, scrolling list, etc...). However, no animation is available. For instance, the companion piece for moving tokens along the arcs does not exist. Instead, symbolic representations of tokens which use algebraic terms are used.

- *Safeguard of a Session*
  The user can, at any moment, save its current environment or load another one. The safeguard includes, among other things, the reachability graph as it has been developed so far. It keeps the labels between the states. The user can also begin his simulation again with the initial environment, and then all information about states and transitions is removed, with the exception of the initial state.

6.2.2 State Graph Explorer

The reachability graph of the system is built during the simulation of the system. When there is no simulation in progress, the states graph can be explored. Various facilities are offered to improve this exploration such as, for instance, a display of all transitions from or to a specific state.

- *Computation of the Successors*
  The environment provides two main means for interaction: a step-by-step evolution and an evolution controlled by expressions of synchronization between the objects in the system. The user can pass indifferently from one type of interaction to the other during a simulation. In a step-by-step evolution the enabling capacities of the object methods are tested in the current state of the system. In a synchronized evolution, a constraint expression involving

events (methods or transitions) of objects in association with notions of sequential order, parallelism and alternatives is specified by the user. The tool tests whether this synchronization expression can emanate from the current state. The test is successful if successors can be computed.

- *Direction of the Development of the Reachability Graph*
  The development of the graph depends on the choice of a particular state from which its successors may be computed. The choice is left to the user or it can be selected randomly by the simulator, the setting of which can be changed during a simulation. From a given state, for instance the current one, the tools proposed its possible output labeled transitions. When once of these transition is pointed, the source and target states are displayed such that the user can study their effect. When the choice of the transition is fixed, the successors will be computed from their target state. The choice of the state from which the reachability graph is developed is totally free, i.e. it can be chosen from the set states which have been already computed. Thus, one development in a branch of the graph can be abandoned for the benefit of another.

6.2.3 Abstract Data Type Checker

During simulation, the user can also test the behavior of the abstract data type part of the system. His task is facilitated by the possibility of defining variables.

- *Variables*
  Various types of variables can be defined. Ground variables, which can be considered as short-cuts of term expressions, free variables and variables defined by an expression which also uses free or ground variables.

- *Evaluation of an Algebraic Term*
  The user can evaluate a term expression in which variables may occur. Equality between two expressions can also be proved by using resolution techniques. When free variables appear in terms if necessary they are instantiated, so as to be as loose as possible. Roughly speaking, the evaluation is performed by means of the orientation of the axioms which describe the behavior of the operations, i.e. the evaluation of an expression can be seen as being a result of its rewriting by means of the axioms.

6.3 Introduction Manual

6.3.1 Organization and Main Windows

- *The Main Windows*
  The simulator and the user manage different windows. The root window allows for the setting of the parameters of the simulation, the specification of the visibility items and is also used to activate the parts of the tool.

- *Displayed Windows*
  Each object is associated with a specific window in which the current marking is displayed. During a step-by-step simulation a window is also associated with each object in which the enabled methods from the current state are displayed and proposed to the user.

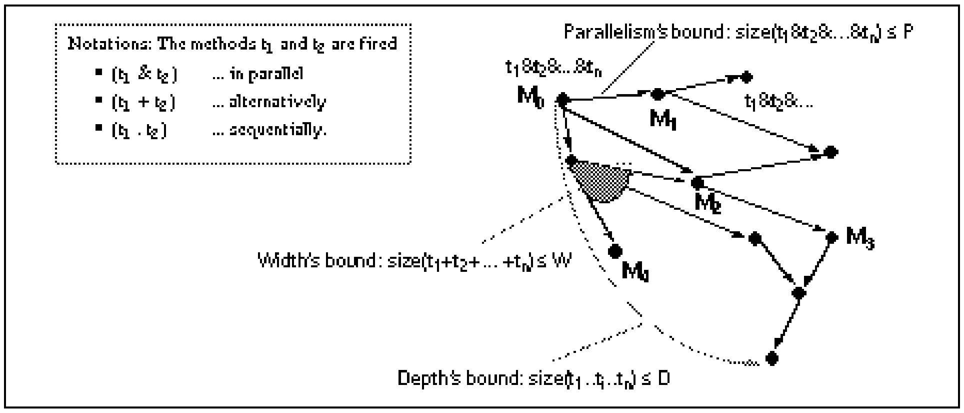

Notations: The methods t1 and t2 are fired
(t1 & t2) ... in parallel
(t1 + t2) ... alternatively
(t1 . t2) ... sequentially.
Parallelism's bound: size(t1&t2&...&tn) ≤ P
t1&t2&...&tn
M0
M1
t1&t2&...
M2
M3
Width's bound: size(t1+t2+... +tn) ≤ W
Mi
Depth's bound: size(t1..ti..tn) ≤ D

Figure 6.2: Bounds Associated with a Synchronization Evaluation

- *The Graph Explorer Window*
  The labeled transitions are displayed in this window and are under the control of settings relative to sources and target states wanted. The use of this window is twofold. On one hand, it is used to show the reachability graph thus far established by the simulation. On the other hand, it allows the user to select successors when a simulation is performed with synchronizations expressions.

6.3.2 Tuning of the Settings and Possible Errors

- *Development of the Reachability Graph*
  In some cases, a synchronization expression may produce a very large or infinite number of successors. If needed, some constraints may be specified in order to overcome this difficulty. Parameters such as depth, parallels and width set a limit to, respectively, the number of steps to take from the current state, the number of transition to fire simultaneously in a single step, and the number of successors of in marking (cf. Figure 6.2).

  The following explanations are relative to a specified marking (M_0 in Figure 6.2) from which its successors are computed during the simulation of one step in the simulator. This computation is constrained as follows :

  1. Two generic markings M_i and M_j linked by an arrow labeled with an expression "$t_1 \,\&\, \ldots \,\&\, t_n$" mean that M_j has been reached from M_i by the simultaneous firing of the transitions $t_1 \ldots t_n$. The parallelism bound P imposes a limit to the number of those transitions which can be fired simultaneously.
  2. The width's bound imposes a limitation to the number of direct successors a marking can have.
  3. At least, the depth's bound set a limit to the longest path from the initial specific marking to its possibly indirect successors computed during the step simulation.

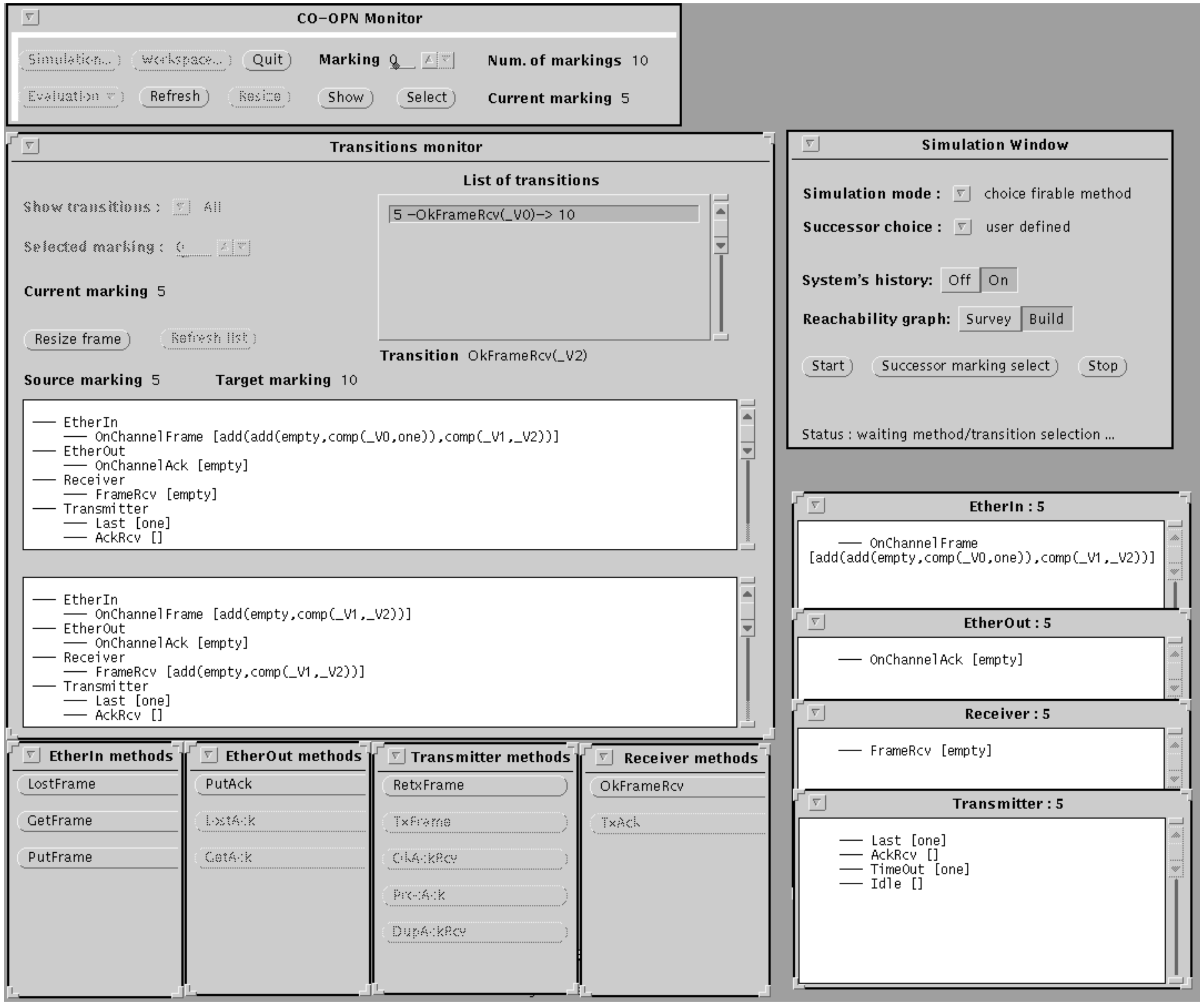
- *Evaluation of the Expression*
  Due to overloading and the use of variables, an expression can yield different evaluations. For a similar reason, a boundary may be established for limiting the number of evaluations.

Moreover, since the definition of a variable expression can, itself, utilize other variable definitions, the user should take care to avoid using cyclic definitions, so that an infinite recursion will not occur.

- *Loading an Environment*
  Some simple tests may be made when a new environment is loaded in order to ensure that it corresponds to the system currently under simulation.

6.4 Miscellaneous

- Implemented in Prolog (ProLog by BIM, license).
- Some of the windows of SANDS$_{1.5}$: cf. Figure 6.3



Marking 0
Num. of markings 10
Current marking 5

Show transitions : All
Selected marking : 0
Current marking 5
Source marking 5
Target marking 10

5 -OkFrameRcv(_V0)-> 10

Transition OkFrameRcv(_V2)

```
── EtherIn
    ── OnChannelFrame [add(add(empty,comp(_V0,one)),comp(_V1,_V2))]
── EtherOut
    ── OnChannelAck [empty]
── Receiver
    ── FrameRcv [empty]
── Transmitter
    ── Last [one]
    ── AckRcv []
```

```
── EtherIn
    ── OnChannelFrame [add(empty,comp(_V1,_V2))]
── EtherOut
    ── OnChannelAck [empty]
── Receiver
    ── FrameRcv [add(empty,comp(_V1,_V2))]
── Transmitter
    ── Last [one]
    ── AckRcv []
```

Simulation mode : choice firable method
Successor choice : user defined
System's history: Off [✓ On]
Reachability graph: Survey [✓ Build]

Status : waiting method/transition selection ...

EtherIn : 5

```
    ── OnChannelFrame
[add(add(empty,comp(_V0,one)),comp(_V1,_V2))]
```

EtherOut : 5

```
    ── OnChannelAck [empty]
```

Receiver : 5

```
    ── FrameRcv [empty]
```

Transmitter : 5

```
    ── Last [one]
    ── AckRcv []
    ── TimeOut [one]
    ── Idle []
```

Figure 6.3: Some of SANDS' Windows

7 The Verifier

Pascal Racloz

7.1 Introduction

This tool is affiliated with a method we developed for addressing the model-checking problem for Petri nets. It is based on a definition of a symbolic representation of sets of markings, and the intended properties of the net are expressed using a specific temporal logic. On one hand, the symbolic representation may structure the state space of the net, to thus allow the consideration of larger systems, with the temporal properties expressed in the intuitive temporal language.

On one hand, the purpose of the model-checking method is to check whether a given temporal property is satisfied by the Petri net modeling and, on the other hand, the definition and the use of a symbolic representation is to cope with the states explosion problem.

7.2 Functional Description

7.2.1 Generalities

- *Petri nets considered*
  Usual nets with some extensions to algebraic nets.

- *Temporal logic*
  The temporal logics allows for expressing qualitative properties related to the arrangement of the system's events during time. Through the temporal constraints on events, these properties characterize how the system evolves. The typical expressible properties are potentiality, invariance and precedence.

- *Predicates structures*
  Our symbolic representation, called the *predicate structure*, has two components which impose some constraints on the set of markings represented. Both the components are symbolic representations of the set of markings which must either belong to or be excluded from the set of markings represented by the predicate structure. A temporal property is represented by a set of predicate structures. In other words, the states verifying the property belong to the states represented by the set of predicates structures.

- *Safeguard*
  At any moment the user can save or load an environment made of predicate structures.

7.2.2 Management of the Predicate Structures

- *Definition of a predicate structures*
  There are two ways of defining a predicate structure. The first one uses a marking of the reachability graph while the second uses the input marking of transition. Both methods generate a predicate structure where the used marking will be the only member of the first component, with the second component left empty. During a session, these markings are kept in order and reused, if needed.

- *Inspection of predicates structures*
  The basic item which can be displayed at once is a marking. The user examines a set of predicate structures by selecting one among them. He chooses a particular predicate structure and can then display the markings belonging the components of the predicate structure.

- *Manipulation of a set of predicate structures*
  A set of predicate structures is identified by a name. Manipulation includes the usual procedures such as *copy* and *delete*, which can bear on an entire set or on particular predicate structure of a given set. Similar operations are also available on the markings of the components a of selected predicate structure. An other class includes operations on sets such as *union*, *intersection* and *negation* of (a) set(s) of predicate structures. Finally an operation of simplification is available, known as *purge*.

7.2.3 Temporal Operators

The temporal operators are those of the CTL temporal logic, namely **EX**, **AX**, **EF**, **AF**, **EG**, **AG**, **E.U.** and **A.U.**. These operators bear on a set of predicate structures and produce a new set. Depending on the temporal operator, the computation of the set sometime involves an iteration process. The user can choose to examine the result of each iteration process or let the computation go on until its completion.

7.3 Introduction Manual

7.3.1 Organization and Main Windows

The temporal tool is activated through the main window of the simulator tool. It is made of two windows, where the first one includes the definitions and the manipulation of the predicate structures and where the other applies the temporal operator.

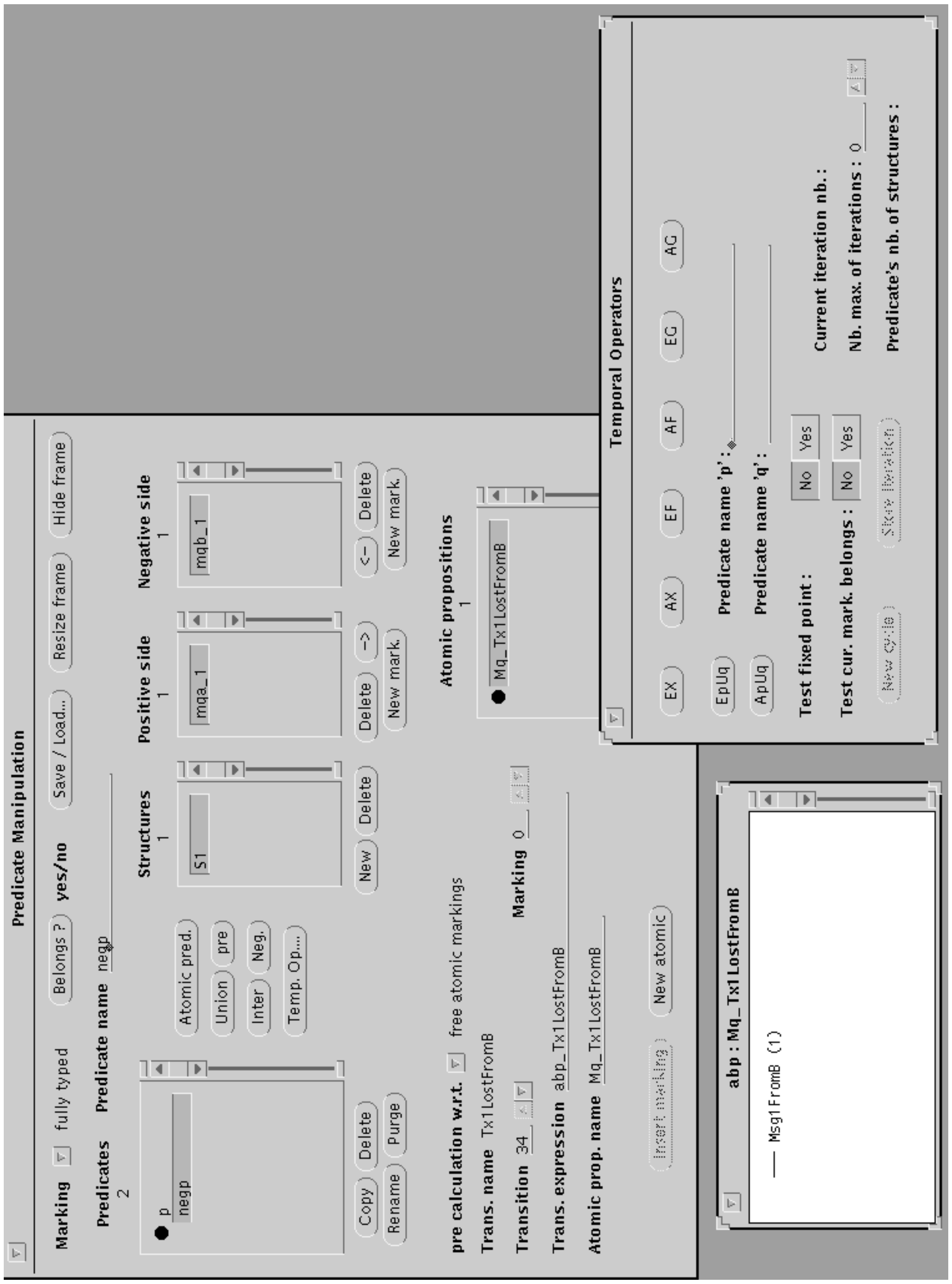
7.3.2 Settings and Errors

Since a set of predicate structures is identified by a name, there is a risk that many errors may occur, such as renaming or copying a set of predicate structures to an already existing name. These errors may be detected and the user may be notified.

Some computations are based on iteration until a fixed-point is reached. Such computations can be very large and even endless. This is the reason why the user is allowed to define a *step* i.e. the number of successive iterations. At the end of a step, if the fixed-point is not reached, a new step can be activated, the result of the step can be saved or the length of the step can be changed.

7.4 Miscellaneous

- Implemented in Prolog (ProLog by BIM). At this moment, a ‘C’ version is under development.
- Some of the temporal tool’s windows: cf. Figure 7.1



Marking fully typed

Predicates 2
- ● p
- negp

Predicate name negp

Structures 1: S1

Positive side 1: mqa_1

Negative side 1: mqb_1

pre calculation w.r.t. free atomic markings

Trans. name Tx1LostFromB

Transition 34

Marking 0

Trans. expression abp_Tx1LostFromB

Atomic prop. name Mq_Tx1LostFromB

Atomic propositions 1
- ● Mq_Tx1LostFromB

Predicate name 'p' :

Predicate name 'q' :

Test fixed point : No Yes

Test cur. mark. belongs : No Yes

Current iteration nb. :

Nb. max. of iterations : 0

Predicate's nb. of structures :

abp : Mq_Tx1LostFromB

— Msg1FromB (1)

Figure 7.1: Some of the Temporal Tool's Windows

8 The Transformer

Mathieu Buffo

8.1 Introduction

The Transformer -TTool- is a tool which applies the TSPP technique (Transformations de Spécifications de problèmes orientées vers le Parallélisme et le Probabilisme [BURB94]) in order to obtain parallel or distributed solutions from a formally specified problem. This tool lets the user manage the gradual transition between the specification and its implementation by the control of each transformation step.

The typical use of TTool consists in the loading of an abstract specification, the iteration of transformation steps preserving the equivalence with the original specification, and the saving of the newly created solution.

8.2 Functional Description

8.2.1 TSPP Framework

Our aim is to derive a parallel solution when starting with a formal description of a problem and using so-called transformations. The framework involves a certain background in several domains, such as specifications, algorithmic, formal methods and parallel architectures.

Specifications

The specification language chosen for the input of the TTool is CO-OPN$_{1.5}$. However, it must be noticed that the TSPP technique is more general and can by applied to a large family of specification languages.

Algorithmic

As the initial abstract specification does not contain algorithmic aspects,these must be introduced into the specification during the derivation of the concrete specification. In other words, the transformation phase is responsible for the search and the choice of the resulting algorithms. This phase is performed in accordance with the methodology for solving the initial problem and with the hardware architecture of the execution platform. At each step, a set of transformations is proposed which use different kinds of algorithms to solve the current problem with the chosen hardware model. The choice and the application of a transformation step then has the effect of introducing its corresponding algorithm into the current specification. Therefore the same problem derived for different machines can lead to different algorithms.

We have developed transformations covering the following algorithmic principles :

- branch and bound,
- divide to conquer,
- dynamic programming,
- searches in graphs (dfs, bfs, simple backtracking, randomized searches, best fit, ...),
- simplification (binary search, tree pruning, ...),
- approximations.

Formal Methods

In order to assure that of our method is safe, we pay great attention to its formal aspects. This notion implies the complete and formal study of all theoretical aspects which deal with transformations of specifications in addition to the choice of a formal specification language and the definition of a suitable equivalence criteria. Before including new concepts into our technique, the relationship to the previous work must be given a sound, formal study. Currently, the theory of temporal logic is used for proving properties of specifications. The theories of computability and complexity are used to derive proper algorithms. Finally, the theory of rewriting systems is used in the transformation management.

Parallel Architectures

In order to derive suitable algorithms from the initial specification, parallel architecture plays a key role. The efficiency of an algorithm is strongly related to the chosen execution platform. To be as general as possible, we introduce into our technique parallel hardware models covering both the theoretical paradigms and real descriptions of machines.

8.2.2 TSPP Technique

The derivation of the solution will be achieved using iterations of transformation steps, as shown in figure 8.1. Each step consists of a simple rewriting of a part of the specification. These steps are performed under certain conditions on the resulting specification, including the equivalence with respect to the initial specification, the adequacy to the concrete execution platform, and the chosen algorithmic design.

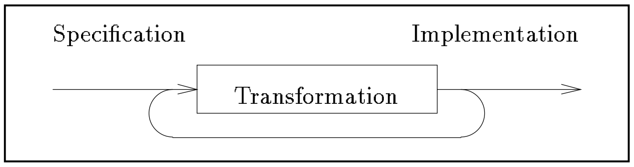


Figure 8.1: Transformation scheme.

An equivalence criteria between two specifications is the main condition for applying a transformation step. If this condition is satisfied, then the result is said to be correct. The choice of such equivalence criteria is crucial if good results are to be obtained. Overly restrictive equivalences will induce badly implemented solutions, and overly permissive ones can introduce significant differences in the obtained behavior. In reality, we use an equivalence notion that belongs to the family of the observational equivalence. The adequacy to the chosen execution platform is realized semi-automatically by the choice of a hardware model which can represent real machine structures as well as theoretical models. The condition is then evaluated using suitable predicates, which return the adequacy of the transformation now based upon the hardware model. The initial specification, produced by the problem analysis, does not contain any algorithmic descriptions. Such algorithms are to be introduced during the transformation steps. According to the initial specification and to the hardware model, we must deduce a suitable algorithmic construction, which will introduce a family of solutions which correspond to the original problem. The refinement of the solution, that is the iteration of the transformation steps, then restricts the set of possible final solutions. Of course, the chosen algorithmic design must be as efficient as possible in order to respect our wishes. Using such a development scheme, if the nature of the problem supports parallelism, we are assured of obtaining a parallel solution.

The transformations are based upon the assumption of the existence of :

- S the set of specifications
- $\approx$ an equivalence relation on $S \times S$

Let T and T_c be two sets, $T = S \times S$ and $T_c \subseteq T$ such that $\langle a, b\rangle \in T_c \Leftrightarrow a \approx b$, where $a, b \in S$. T is called the set of transformations, and an element $t \in T$ is a *transformation*. A transformation $t \in T_c$ is said to be *correct*. The application of the transformation $\langle a, b\rangle \in T$ is the transformation step passing from the specification a to the specification b.

The *transformation rules* define generic ways of transforming specifications. Their semantics are based on rewrite systems. Two semantics are associated with each transformation rule :

- the "test" semantics, which allows the testing of the application of the rule, with respect to the current specification and the initial data (this data controls the desired effects),
- the "apply" semantics, which derive a valid transformation step from the previous tested rule and initial data, and apply it.

A rule R induces a family of transformations $T_R \subseteq T$, defined by (for $t = \langle a, b\rangle \in T$) :
$t \in T_R \quad \Leftrightarrow \quad \exists I$ (initial data) such that $Test_R(a, I)$ is true and b is the result of $apply_R(a, I)$
A rule is said to be correct if $T_R \subseteq T_c$, that is if all derived transformations are correct. The rules are the simplest way of describing transformations with our technique.

The *transformation methods* allow for a generic description of a sequence of transformation steps. Their semantics are based on Petri nets. As for the transformation rules, two semantics are associated with each transformation method: the "test" and the "apply" semantics. Similarly, a transformation method M induces a family of transformations $T_M \subseteq T$, $(t = \langle a, b\rangle \in T)$:
$t \in T_M \quad \Leftrightarrow \quad \exists I$ (initial data) such that $Test_M(a, I)$ is true and b is the result of $apply_M(a, I)$
These semantics are based on Petri nets, as has been said above. The application of a method can be considered as the firing of certain transitions of such a net, with each of these transitions invoking a simple transformation step. The transformation methods are useful for grouping simple transformations into those which are more complex, with a synthetic approach.

8.2.3 TTool

The TTool - Transformation Tool package - was developed for performing transformations on specifications with the TSPP technique. The basic idea of the transformation tool is an adequate rewrite system which allows for derivations of new specifications written in CO-OPN$_{1.5}$ from those previous, under the condition of an observational equivalence of these specifications, which guarantees the correctness of the resulting parallel specification at the implementation level.

8.3 Introduction Manual

The transformation tool package consists of the tool itself, which performs the transformation, and the transformation library, which is used by the tool and which can be modified dynamically.

8.3.1 The Tool

The tool is a user-friendly xview-based application. The refinement is performed step-by-step, according to the basic procedures of the transformation technique. At each step, the user can :

- decide to stop the transformation process,
- backtrack, if the specification developed is not satisfactory,
- evaluate the adaptation of the current specification to the execution platform,
- decide to perform a new transformation on the current specification.

The transformation itself is very simple to manage. The user selects the correct transformation rule or a transformation method to apply and TTool then displays the possible transformation steps derived from the selected rule or method (using the "test" semantics). The user can then choose one of these steps, and TTool will subsequently apply it, using the "apply" semantics.

8.3.2 The Library

The library is composed of several textual modules, written in a CO-OPN$_{1.5}$-like syntax. These modules are split into two classes, according to the TSPP technique :

- The rule modules, which contain transformation rules and are similar to the algebraic modules in CO-OPN, due to of the similarity of the rewriting rules and the algebraic operations.
- The method modules, which each contain a transformation method and are similar to the object modules in CO-OPN, because all are, in fact, Petri net descriptions. By imposition, all transformation methods must be correct.

9 The ADA Translator N/A

Christophe Buffard

9.1 Introduction

Ada offers several mechanisms for expressing concurrency, namely tasks and protected types. But, concurrency can introduce significant problems which are inherent in the program's interactions. Amongst these problems or necessary properties, we can mention deadlocks, fairness and particular temporal properties. The modeling of concurrent behavior with tools can help prevent these problems. For this purpose, we show in this part that programs written in Ada can be modeled using the formalism CO-OPN$_{1.5}$, based on Petri nets and algebraic specifications, which offers the possibility of selecting the level of abstraction of the modeling, that to say, which part must be modeled using Petri nets or algebraic abstract data types. These modelings can be used in the detection of the program anomalies. We present here an internal working description of the Ada translator

9.2 Functional Description

9.2.1 Construction of CO-OPN$_{1.5}$ Model

We can represent the Ada code in the CO-OPN$_{1.5}$ model with varying levels of abstraction, depending upon the point of view adopted. For example, we can stay very close to the code or take a much more abstract view. A program is a sequence of statements and only the necessary statements will be considered in the model. The level of abstraction taken will depend on our needs, with the principal objective being to detect errors such as deadlocks ... . Our initial approach is to model intertask communication while conserving the structures of the exchanged data. Data are modeled with the necessary abstract level depending on the point of view of the analysis [BB94]. The novelty of our approach, in comparison to others in this domain, is modeling both the communication and the data structures involved. The correspondences between Ada data structure and the CO-OPN$_{1.5}$ component are shown in Figure 9.1. For intertask communication we have retained two categories of streams. The first stream, which can alter the control flow, is defined by Ada reserved words (**if**, **loop**, **requeue** and **select**), while the second constitutes the rendez-vous, defined by Ada reserved word (**accept**) and the *entry calls.*

As we have stated above, the main goal is to anticipate errors due to concurrency in programs. We choose to retain only the functions and procedures that are directly linked with statements expressing concurrency in Ada. We have two ways to model the function or procedure:

Ada	CO-OPN$_{1.5}$	
	Petri Nets	ADT
ADT Inclusion of Types		●
Procedures	●	
Tasks Protected Types	●	
Functions (pure)		●
Exceptions	?	?

Figure 9.1: Correspondence between Ada Components and CO-OPN$_{1.5}$

- a function or procedure, calling a task or accepting an entry call, is modeled with a Petri net, or
- it is modeled with the CO-OPN$_{1.5}$ object method.

In this way, we keep only the necessary representation of concurrence and data. Tasks are modeled with a CO-OPN$_{1.5}$ object, as we have seen above, and the task's functions are modeled in the same way as other functions. Protected types (passive tasks) are modeled like tasks. The requeue statement can be use to complete an accept statement or entry body, while redirecting the corresponding entry call to a new entry queue. The requeue statement is modeled in CO-OPN$_{1.5}$ by an ADT given in the description of the Fifo. It is also used to model the waiting line of a task entry. An exception is a reaction to an exceptional or unusual situation, or what we sometimes call an error. The exception have not yet been tackled.

9.3 Scheme of the Construction

In this section we give an example of how an Ada rendez-vous and protected types are handled within our framework. The package module can be included in both parts of the CO-OPN$_{1.5}$ model. When a package contains a type description, it is owned by the ADT part of the model. Otherwise, when a package contains a procedure, function or task body, it is represented with Petri nets. This correspondence is depicted in Figure 9.2.

9.3.1 Ada Tasks

Ada offers several mechanisms for expressing concurrency: tasks, which are active entities, and protected types, which can be considered as a passive tasks. Tasks express concurrency and their execution can take place on one or more processors. Tasks allow simultaneous execution and the order of execution is undefined except when two tasks participate in a rendez-vous, which is the

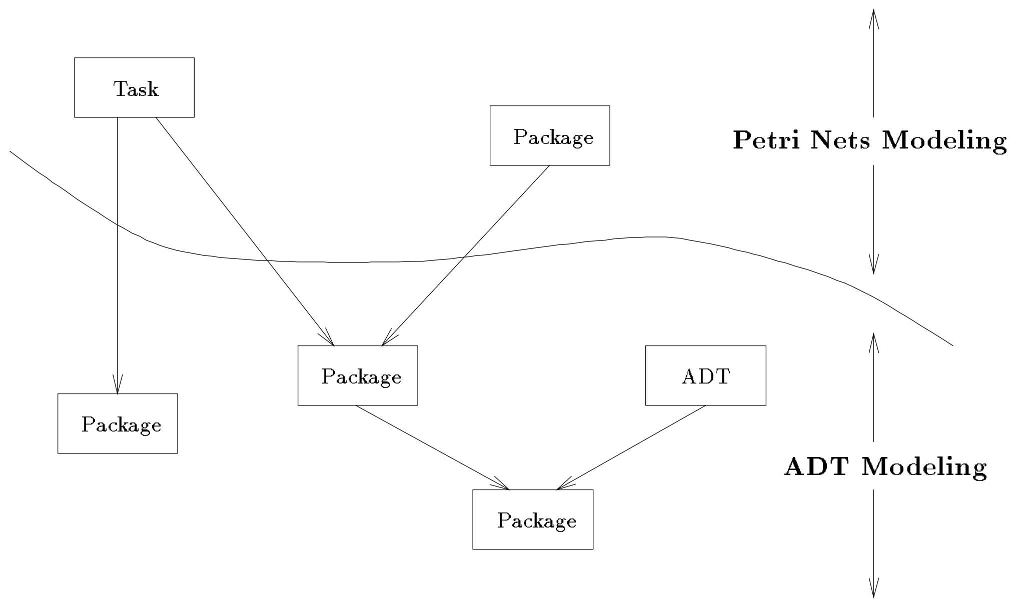


Figure 9.2: Ada Entities Representation

only form of direct communication between tasks. A rendez-vous is asymmetric: the caller must know the called task while the called task is unaware of the identity of the caller. Calling an entry of a task is very similar to calling a procedure with the exception that the called task can decide when and how to accept the call. If the called task is not ready to accept the entry, then the execution of the calling task is suspended until the called task is ready to accept the entry call. Once the called task accepts the entry, the rendez-vous begins. The duration of the rendez-vous is controlled by the called task, and is equal to the time necessary to execute the code contained in the accept statement. Ada specifies a model of concurrency based on the asynchronous execution of each task as if each task had its own processor. In contrast to the asynchronous model of execution, communication between two tasks takes place through a rendez-vous, which is in synchrony. The general mechanism for the cooperation between tasks is:

- either based on message passing, where a task communicates with another by sending a message through the tasking kernel,
- or based on the explicit use of synchronization.

9.3.2 Protected Types

Ada language specified protected types and they formalize and expand on the use of passive tasks of Ada 83. Protected types offer the possibility of defining a passive object, such as a monitor. A protected type consists of functions, procedures and entries which are mutually exclusive. We can also define entry, which is different from the functions or procedures, due to the fact that entry defines a barrier condition.

Protected types are modeled by a CO-OPN$_{1.5}$ object. The objects' methods represent the way of calling a function. There is also a mechanism for insuring the mutual exclusion of the different functions that compose the protected type.

9.4 Model Analysis

The analysis of the system can be roughly divided into two cases: the verification of whether the system is bounded or not (in terms of the amount of data needed for its evolution) and whether parts of the system or the entire system have the possibility to evolve. The checking of these properties, well-developed for the usual Petri nets, is difficult in our framework because of the use of the structured tokens and the modularity. Presently, the means we investigate cover the following techniques. In some cases, from the underlying structure of the Petri nets (projection of the data to unstructured tokens) some properties can be deduced. The simulation [BFR93a] is very useful for giving a first impression about the intended behavior of the system. Finally, we study the use of temporal logic as a tool for the specification and verification of the system in terms of the occurrence of its events over the course of the time [BG91, RB93].

9.5 Perspectives

We have just presented an overview of the Ada code model. The advantage of this model is that it allows us both to model the data structures and to preserve the oriented object paradigm across CO-OPN$_{1.5}$. Much work is yet to be done in order to model all of the Ada code, in particular, the exceptions statement that we have not yet tackled. The way Ada 95 components are represented, either by Petri nets or algebraic specifications, shown in Figure 9.1, is still subject to modifications. Up to now, this correspondence is intuitive, dynamic items are handled by Petri nets and data structures by ADT, but this correspondence can be reconsidered by future examples. Our efforts bear also upon extensions of the verification techniques to the entire CO-OPN$_{1.5}$ framework.

10 The MIMD Compiler

Jarle Hulaas & Mathieu Buffo

10.1 Introduction

CO-OPN$_{1.5}$ specifications can be either simulated using a monitor or executed on a MIMD machine. In both instances, the CO-OPN$_{1.5}$ code must to be processed by a common front-end which performs lexical, syntactical and static semantic analysis.

Then, in the case where a simulation is desired, the specifications are translated into Prolog. The resulting code can be loaded into the monitor and the simulator for a highly interactive evaluation of the application.

If the user needs a more efficient execution of his specifications, he can take advantage of the MIMD compiler, which generates either parallel C or standard C with the PVM environment, according to the target platform: the Transputer hypercube Volvox Archipel, or a MIMD machine with PVM (Parallel Virtual Machine) [GBD$^+$93] message passing, ranging from a network of workstations to the Cray T3D supercomputer. The compiler accepts only a subset of the CO-OPN$_{1.5}$ language. Some simple rules concerning the level of abstractness of the specifications have been established in order to gain maximal performance during their execution. The Transformation Tool, described in chapter 8, may help the abstract specifications evolve into lower-level expressions.

10.2 How the Compiler Works

CO-OPN$_{1.5}$ specifications are split into two kinds of modules. The first are the algebraic specification modules, which describe the abstract data types used. These are translated into the functional language OPAL , and then the OPAL system compiles them into C code. The second kind of modules are the Petri net objects, which are directly compiled into the C dialect appropriate for the target architecture.

These steps are executed transparently to the user, who then simply has to launch the MIMD compiler by submitting the main module of the system. The result is an executable file, completed on the Volvox by a mapping of the tasks onto the processor network, as well as a shell script that activates the application. On a PVM system the configuration is easier to set up, and it is thus not fixed at compile time.

For convenience, the most frequently used data types (boolean and integer) are predefined in the compiler.

10.3 Description of the Accepted Source Language

The MIMD compiler can process a subset of the original CO-OPN$_{1.5}$ language. This restriction is essential for reasons of efficiency. The expressiveness of CO-OPN$_{1.5}$ is such that a logic interpreter is required to exploit it completely. The subsequent conditions are stated:

- Only simple modules are compiled, excluding genericity.
- In algebraic modules, axioms are limited to the form of conditional rewrite rules, where the left-hand term of an equation is made of ground expressions. Only simple variables or constant generators are allowed as operation parameters. In other words, operations must be defined such that the pattern-matching of their arguments is reduced to comparisons with constants.
- In object modules, the unification mechanism that acts on abstract data values is replaced by testing or assignment, according to the particular conditions:
  1. If we are calculating a postcondition, assignment is chosen.
  2. In a condition part, testing is preferred.
  3. In a general case, if the algebraic term reduces to a variable, then assignment is adopted. Otherwise testing is implemented.
- As a consequence, an order must be introduced into the evaluation of a transition, and it is the following:
  1. The preconditions.
  2. The synchronization.
  3. The conditions.
  4. The postconditions.
- Methods are fired only when an object synchronizes with them.

We are currently working on extending the subset of the specification language that can be compiled efficiently. For instance, filtering could easily also process parameterized generators instead of only constants.

10.4 Termination of a Program

The notion of termination is not defined in CO-OPN$_{1.5}$. Therefore, the following convention has been taken: A system is considered as terminated when the main object(s) is (are) in a stable state.

10.5 Input and Output

The CO-OPN$_{1.5}$ language has a construction for the definition of the initial marking of a Petri net. A more flexible approach is given by the compiler, through a command-line option which indicates the name of the places for which the user must provide the contents at the moment of program startup. For this purpose, a routine (written in C) must exist for each type concerned. These routines are first searched for by the compiler in the current directory, and then in the directories specified by CO-OPN$_{1.5}$ environment variables.

Output is governed by the same kind of mechanism. A compiler option lists the places in which the contents must be printed at program termination.

10.6 Perspectives

Current work on the MIMD compiler is directed towards developing a prototyping tool and method adapted to parallel applications. In particular, we wish to provide the user with an opportunity of mixing, at any level of the software life-cycle, tool-generated compilation units with a hand-written code. This process is known as mixed prototyping [CK90]. The expected benefits are multiple: higher efficiency, more flexible user interface, instant realization of integration tests, and an ideal support for systematic refinement schemes. Such research will lead to converting the compiler to a C++ or Ada95 code-generator.

10.7 Auxiliary Tools

The CO-OPN$_{1.5}$ MIMD compiler is not bound to any OS or windowing system. The following tools are all it needs to function:

- For compiling and linking
  - Lex & Yacc
  - ANSI C compiler
  - PVM 3.0 library (http://www.epm.ornl.gov/pvm/)
  - OPAL 2.1 compiler and library (http://www.cs.tu-berlin.de/ opal/)
- For execution
  - PVM V3.0 (the console program as well as the daemon)
  - OPAL 2.1 library
- Monitoring tools (optional)
  - On Volvox: the 'ParaGraph' visualization tool works on tracefiles generated by executing specifications compiled with the '-m' option.
  - With PVM: 'XPVM' is a graphical interface for PVM and requires no particular compiler options. The monitoring is done directly during program execution.

A Basic ADTs

O. Biberstein & D. Buchs & G. Di Marzo

```
(:-------------------------------------------------------------------------*
| Specifications of standard basic ADT  'basictypes'.                      |
|                                                                          |
| Several abstract data types are given in this standard package,          |
| they define the most usefull types that we generally use.                |
| Part of these types are basic values while others are type               |
| constructors.                                                            |
|                                                                          |
| Semantics: The semantics used is based on the implicit definition of     |
| the validity domain of the functions by the axioms. The possible         |
| values are determined by the generators (finitely generated with         |
| respect to the generators). A function is not defined if it has no       |
| axiom for particular value.                                              |
|                                                                          |
| Provided modules;                                                        |
|                                                                          |
|  Basic types:                                                            |
|   Unary, Booleans, Naturals, Characters.                                 |
|                                                                          |
| Authors : D. Buchs previous versions from O.Biberstein and G.di Marzo    |
| Date    : 5 July 1995,                                                   |
| Revised : O. Biberstein                                                  |
| Date    : 19 Oct 1995                                                    |
*-------------------------------------------------------------------------:)


(:-------------------------------------------------------------------------*
| Specification of a unique value.                                         |
|                                                                          |
| The unique type                                                          |
|                                                                          |
| Author : D. Buchs                                                        |
| Date   : 3 July 1995                                                     |
*-------------------------------------------------------------------------:)

Adt Unique;
Interface
  Sort unique;
  Generator
    @ : -> unique;
Body
End Unique;
```

```
(:------------------------------------------------------------------------*
| Specification of the booleans.                                          |
|                                                                         |
| The boolean type with true, false generators, usual logical             |
| operations (not, and, or, xor) and equal operation is defined.          |
|                                                                         |
| Author  : O. Biberstein                                                 |
| Date    : 29 March 93                                                   |
| Revised : G. Di Marzo                                                   |
| Date    : 15 June 1995                                                  |
| Revised : O. Biberstein                                                 |
| Date    : 95/10/19                                                      |
*------------------------------------------------------------------------:)

Adt Booleans;
Interface
  Sort boolean;
  Generators
    true    : -> boolean;
    false   : -> boolean;
  Operations
    not   _ : boolean -> boolean;
    _ and _ : boolean boolean -> boolean;
    _ or  _ : boolean boolean -> boolean;
    _ xor _ : boolean boolean -> boolean;
    _ = _   : boolean boolean -> boolean;
Body
  Axioms
    not true      = false;
    not false     = true;

    true  and b   = b;
    false and b   = false;

    true  or b    = true;
    false or b    = b;

    false xor b   = b;
    true  xor b   = not b;

    (true=true)   = true;
    (true=false)  = false;
    (false=true)  = false;
    (false=false) = true;
  Where
    b : boolean;
End Booleans;
```

```
(:----------------------------------------------------------------------*
| Natural numbers.                                                      |
|                                                                       |
| Naturals type is defined over generators 0 and succ.                  |
| Arithmetic, relational and constant (Nx) are provided. In case of     |
| division by 0 or if y > x in (x-y) the n operations give 0.           |
|                                                                       |
| Author  : O. Biberstein                                               |
| Date    : 29 March 93                                                 |
| Revised : G. Di Marzo                                                 |
|           O. Biberstein                                               |
| Date    : 15 June 1995                                                |
|           95/10/19                                                    |
*----------------------------------------------------------------------:)

Adt Naturals;
Interface
  Use  Booleans;
  Sort natural;
  Generators
    0       : -> natural;
    succ _  : natural -> natural;
  Operations
    _ + _   ,
    _ - _   ,
    _ * _   ,
    _ / _   ,
    _ % _   : natural natural -> natural;
    _ = _   ,
    _ <= _  ,
    _ < _   ,
    _ > _   ,
    _ >= _  : natural natural -> boolean;
    max _ _ : natural natural -> natural;
    even _  : natural -> boolean;
    2** _   ,
    _ ** 2  : natural -> natural;
    ;; constants
    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20 : -> natural;
Body
  Axioms
    0+x = x;
    (succ x)+y = succ (x+y);
    ;; substraction, if y > x then x-y = 0
    0-x = 0;
    (succ y)-0 = succ y;
    (succ y)-succ x = y-x;
    0*x = 0;
    (succ x)*y = (x*y)+y;
    ;; division, if y = 0 then div x y = 0
    x/0 = 0;
    x<y  = true => x/y = 0;
    x>=y = true => x/y = succ ((x-y)/y);
    ;; modulo, if y = 0 then mod x y = 0
    x%y = x-(y*(x/y));
    0=0             = true;
    0=succ x        = false;
    succ x=0        = false;
    (succ x)=succ y = x=y;
    x<=y = not y<x;
    0<0             = false;
    0<succ x        = true;
    succ x < 0      = false;
    succ x < succ y = x<y;
    x>y = not x<=y;
    x>=y = not x<y;
    even 0 = true;
    even succ x = not even x;
    2**0 = succ 0;
    2**succ x = (succ succ 0)*(2**x) ;
    (x>=y)=true  => max x y = x ;
    (x>=y)=false => max x y = y ;
    x**2 = x*x;
    1  = succ 0;   2  = succ 1;   3  = succ 2;   4  = succ 3;
    5  = succ 4;   6  = succ 5;   7  = succ 6;   8  = succ 7;
    9  = succ 8;   10 = succ 9;   11 = succ 10;  12 = succ 11;
    13 = succ 12;  14 = succ 13;  15 = succ 14;  16 = succ 15;
    17 = succ 16;  18 = succ 17;  19 = succ 18;  20 = succ 19;
  Where
    x, y : natural;
End Naturals;
```

```
(:-----------------------------------------------------------------------*
| Integer numbers.                                                      |
|                                                                       |
| Integers type is defined over generators 0, pred, succ.              |
| Arithmetic, relational and constant (Nx) are provided. In case of    |
| division by 0 or if y > x in (x-y) the n operations give 0.          |
|                                                                       |
| Author  : O. Biberstein                                               |
| Date    : 29 March 93                                                 |
| Revised : G. Di Marzo                                                 |
| Date    : 15 June 1995                                                |
| Revised : D. Buchs                                                    |
| Date    : 20 July 1995                                                |
| Revised : O. Biberstein                                               |
| Date    : 19 Oct 1995                                                 |
*-----------------------------------------------------------------------:)

Adt Integers;
Interface
  Use   Booleans;
  Sort integer;
  Generators
    0       : -> integer;
    succ _  ,
    pred _  : integer -> integer;
  Operations
    - _     : integer -> integer;
    _ + _   ,
    _ - _   ,
    _ * _   ,
    _ / _   ,
    _ % _   : integer integer -> integer;
    _ = _   ,
    _ <= _  ,
    _ < _   ,
    _ > _   ,
    _ >= _  : integer integer -> boolean;
    even _  : integer -> boolean;
    2 ** _  : integer -> integer;
    max _ _ : integer integer -> integer;
    _ ** 2  : integer -> integer;
   ;; constants
   1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20 : -> integer;
Body
  Axioms
    pred succ x = x;
    -0 = 0;
    -succ x = pred -x;
    -pred x = succ -x;
    x+0 = x;
    x+succ y = (succ x)+y;
    x+pred y = (pred x)+y;
    ;; substraction
    x-0 = x;
    x-succ y = (pred x)-y;
    x-pred y = (succ x)-y;
    0*x        = 0;
    (succ x)*y = (x*y)+y;
    (pred x)*y = (x*y)-y;
    ;; division, if y = 0 then div(x,y) = 0 ;;
    (y*d <= x) and (x < y*succ d) = true  => x/y = d;
    (y*d <= x) and (x < y*succ d) = false => x/y = x/y;
    x/0 = 0;
    ;; modulo, if y = 0 then mod(x,y) = 0
    x%y = x-(y*(x/y));
    0<0        = false;
    0<succ 0   = true;
    0<succ y   = true  => 0 < succ succ y = true;
    0<pred y   = false => 0 < pred y = false;
    succ x < y = x < pred y;
    pred x < y = x < succ y;

    x<=y = not y<x;
    x=y  = not (x<y or y<x);
    x>y  = not x<=y;
    x>=y = not x<y;

    even 0 = true;
    even succ x = not even x;
    even pred x = not even x;

    2**0 = 1 ;
    2**succ x = 2*(2**x);
    2**pred x = (2**x)/2;

    (x>=y)=true  => max x y = x ;
    (x>=y)=false => max x y = y;

    x**2 = x*x;

    1  = succ 0;    2 = succ 1;   3  = succ 2;    4 = succ 3;
    5  = succ 4;    6 = succ 5;   7  = succ 6;    8 = succ 7;
    9  = succ 8;   10 = succ 9;   11 = succ 10;  12 = succ 11;
    13 = succ 12;  14 = succ 13;  15 = succ 14;  16 = succ 15;
    17 = succ 16;  18 = succ 17;  19 = succ 18;  20 = succ 19;
  Where
    x, y, d : integer;
End Integers;
```

```
(:------------------------------------------------------------------*
| Characters                                                        |
|                                                                   |
| Characters type is defined over generators all the ASCII character<128|
| code indicates the ASCII value of the character while char        |
| is the opposite.                                                  |
|                                                                   |
| Author  : D. Buchs                                                |
| Revised : O. Biberstein                                           |
| Date    : 5 July 1995                                             |
| Date    : 95/10/20                                                |
*------------------------------------------------------------------:)

Adt Characters;
Interface
  Use   Booleans, Naturals;
  Sorts character;
  Generators
    nul, soh, stx, etx, eot, enq, ack, bel,
    bs , ht , lf , vt , ff , cr , so , si ,
    dle, dc1, dc2, dc3, dc4, nak, syn, etb,
    can, em , sub, esc, fs , gs , rs , us ,
    sp, !, quote, #, $, %, &, ',
    lpar, rpar, *, +, comma, -, dot, /,
    0, 1, 2, 3, 4, 5, 6, 7,
    8, 9, colon, semicolon, <, =, >, ?,
    @, A, B, C, D, E, F, G,
    H, I, J, K, L, M, N, O,
    P, Q, R, S, T, U, V, W,
    X, Y, Z, [, \, ], ^, underline,
    `, a, b, c, d, e, f, g,
    h, i, j, k, l, m, n, o,
    p, q, r, s, t, u, v, w,
    x, y, z, {, |, }, ~, del,
    topofcharacters : -> character;
  Operations
    code      : character -> natural;
    character : natural -> character;
    succ _    ,
    pred _    : character -> character;
    _ < _     ,
    _ = _     : character character -> boolean;
Body
  Axioms
    succ nul = soh;    succ soh = stx;    succ stx = etx;
    succ etx = eot;    succ eot = enq;    succ enq = ack;
    succ ack = bel;    succ bel = bs;     succ bs  = ht;
    succ ht  = lf;     succ lf  = vt;     succ vt  = ff;
    succ ff  = cr;     succ cr  = so;     succ so  = si;
    succ si  = dle;    succ dle = dc1;    succ dc1 = dc2;
    succ dc2 = dc3;    succ dc3 = dc4;    succ dc4 = nak;
    succ nak = syn;    succ syn = etb;    succ etb = can;
    succ can = em;     succ em  = sub;    succ sub = esc;
    succ esc = fs;     succ fs  = gs;     succ gs  = rs;
    succ rs  = us;     succ us  = sp;     succ sp  = !;
    succ !   = quote;  succ quote = #;    succ #   = $;
    succ $   = %;      succ %   = &;      succ &   = ';
    succ '   = lpar;   succ lpar = rpar;  succ rpar = *;
    succ *   = +;      succ +   = comma;  succ comma = -;
    succ -   = dot;    succ dot = /;      succ /   = 0;
    succ 0   = 1 in Characters;           succ 1   = 2 in Characters;
    succ 2   = 3 in Characters;           succ 3   = 4 in Characters;
    succ 4   = 5 in Characters;           succ 5   = 6 in Characters;
    succ 6   = 7 in Characters;           succ 7   = 8 in Characters;
    succ 8   = 9 in Characters;           succ 9   = colon;
    succ colon = semicolon;         succ semicolon = <;
    succ <   = =;
    succ =   = >;      succ >   = ?;      succ ?   = @;
    succ @   = A;      succ A   = B;      succ B   = C;
    succ C   = D;      succ D   = E;      succ E   = F;
    succ F   = G;      succ G   = H;      succ H   = I;
    succ I   = J;      succ J   = K;      succ K   = L;
    succ L   = M;      succ M   = N;      succ N   = O;
    succ O   = P;      succ P   = Q;      succ Q   = R;
    succ R   = S;      succ S   = T;      succ T   = U;
    succ U   = V;      succ V   = W;      succ W   = X;
    succ X   = Y;      succ Y   = Z;      succ Z   = [;
    succ [   = \;      succ \   = ];      succ ]   = ^;
    succ ^   = underline;          succ underline = `;
    succ `   = a;      succ a   = b;      succ b   = c;
    succ c   = d;      succ d   = e;      succ e   = f;
    succ f   = g;      succ g   = h;      succ h   = i;
    succ i   = j;      succ j   = k;      succ k   = l;
    succ l   = m;      succ m   = n;      succ n   = o;
    succ o   = p;      succ p   = q;      succ q   = r;
    succ r   = s;      succ s   = t;      succ t   = u;
    succ u   = v;      succ v   = w;      succ w   = x;
    succ x   = y;      succ y   = z;      succ z   = {;
    succ {   = |;      succ |   = };      succ }   = ~;
    succ ~   = del;
    succ del = topofcharacters;
    succ topofcharacters = topofcharacters;
    !(cc=topofcharacters) => code cc = (code succ cc)-1;
    code topofcharacters = 2**7;   ;; 128
    character (code cc) = cc;
    !(succ cc=nul) => pred succ cc = cc;
    succ cc=nul => pred succ cc = topofcharacters;
    (!(cx=topofcharacters)) & !(cy=topofcharacters) =>
      cx<cy = succ cx < succ cy;
    !(cx=topofcharacters) => cx<topofcharacters = true;
    !(cy=topofcharacters) => topofcharacters<cy = false;
    topofcharacters<topofcharacters = false;
    (!(cx=topofcharacters)) & !(cy=topofcharacters) =>
      cx=cy = (succ cx = succ cy);
    !(cx=topofcharacters) => cx=topofcharacters = false;
    !(cy=topofcharacters) => topofcharacters=cy = false;
    topofcharacters=topofcharacters = true;
  Where
    cc, cx, cy : character;
End Characters;
```

```
;; Standard Library of ADT
;;
;; basictypes.sys

Specification   basictypes;
Version         1.0;
Date            12 September 1995;

Authors D.Buchs : Unary, Booleans, Naturals, Integers, Characters;

Modules Unary, Booleans, Naturals, Integers, Characters: basictypes;

End basictypes;
```

```
(:-----------------------------------------------------------------------*
| Specifications of standard ADT.  'constructortypes'                    |
|                                                                        |
| Several abstract data types are given in this standard package,        |
| they define the most usefull parameters and type constructors.         |
|                                                                        |
| Semantics: The semantics used is based on the implicit definition of   |
| the validity domain of the functions by the axioms. The possible       |
| values are determined by the generators (finitely generated with       |
| respect to the generators). A function is not defined if it has no     |
| axiom for particular values.                                           |
|                                                                        |
| Provided modules:                                                      |
|                                                                        |
| Parameter types:                                                       |
|  Elem, ComparableElem, ComparableElem1, ComparableElem2, OrderedElem.  |
|                                                                        |
| Constructor types:                                                     |
|  List, OrderedList, Pc2, Fifo, Lifo, Set, Bag, Table.                  |
|                                                                        |
| Authors : D. Buchs previous versions from O.Biberstein and G.di Marzo  |
| Date    : 5 July 1995                                                   |
| Revised : O. Biberstein                                                |
| Date    : 19 October 1995                                              |
*-----------------------------------------------------------------------:)


(:-----------------------------------------------------------------------*
| Parameter specification.                                               |
|                                                                        |
| The elem type is defined without operation                             |
|                                                                        |
| Author  : O. Biberstein                                                |
| Date    : 29 March 93                                                  |
| Revised : G. Di Marzo                                                  |
| Date    : 15 June 1995                                                 |
*-----------------------------------------------------------------------:)

Parameter Adt Elem;
Interface
  Sort elem;
Body
End Elem;


(:-----------------------------------------------------------------------*
| Parameter specification.                                               |
|                                                                        |
| The elem type is defined with an equal operation and of course uses   |
| the Booleans module.                                                   |
|                                                                        |
| Author  : O. Biberstein                                                |
| Date    : 29 March 93                                                  |
| Revised : G. Di Marzo                                                  |
| Date    : 15 June 1995                                                 |
*-----------------------------------------------------------------------:)

Parameter Adt ComparableElem;
Interface
  Use  Booleans;
  Sort elem;
  Operation
    _ = _ : elem elem -> boolean;
Body
  Theorems
    ;; usual equivalence relation properties
    ;; reflexivity
    (x = x) = true;
    ;; symmetry
    (x = y) = true => (y = x) = true;
    ;; transitivity
    (x = y) = true & (y = z) = true => (x = z) = true;
  Where
    x, y, z : elem;
End ComparableElem;


(:-----------------------------------------------------------------------*
| Parameter specification.                                               |
|                                                                        |
| The elem type is defined with an equal and an order relation           |
| operation and of course uses                                           |
| the Booleans module.                                                   |
|                                                                        |
| Author  : O. Biberstein                                                |
| Date    : 29 March 93                                                  |
| Revised : G. Di Marzo                                                  |
| Date    : 15 June 1995                                                 |
*-----------------------------------------------------------------------:)

Parameter Adt OrderedElem;
Interface
  Use  Booleans;
  Sort elem;
  Operations
    _ < _ ,
    _ = _ : elem elem -> boolean;
Body
  Theorems
    ;; usual order relation properties
    ;; anti-symmetry
    (x < y) = true & (y < x) = true => (y = x) = true;
    ;; transitivity
    (x < y) = true & (y < z) = true => (x < z) = true;
    ;; usual equivalence relation properties
    ;; reflexivity
    (x = x) = true;
    ;; symmetry
    (x = y) = true => (y = x) = true;
    ;; transitivity
    (x = y) = true & (y = z) = true => (x = z) = true;
  Where
    x, y, z : elem;
End OrderedElem;


(:-----------------------------------------------------------------------*
| Parameter specification.                                               |
|                                                                        |
| The elem1 type is defined with an equal operation and of course uses   |
| the Booleans module.                                                   |
|                                                                        |
| Revised : G. Di Marzo                                                  |
| Date    : 15 June 1995                                                 |
*-----------------------------------------------------------------------:)

Parameter Adt ComparableElem1;
Interface
  Use Booleans;
  Sorts elem1;
  Operations
    _ = _ : elem1 elem1 -> boolean;
Body
  Theorems
    ;; usual equivalence relation properties
    ;; reflexivity
    (x = x) = true;
    ;; symmetry
    (x = y) = true => (y = x) = true;
    ;; transitivity
    (x = y) = true & (y = z) = true => (x = z) = true;
  Where
    x, y, z : elem1;
End ComparableElem1;


(:-------------------------------------------------------------------*
| Parameter specification.                                          |
|                                                                   |
| The elem2 type is defined with an equal operation and of course uses |
| the Booleans module.                                              |
|                                                                   |
| Revised : G. Di Marzo                                             |
| Date    : 15 June 1995                                            |
*-------------------------------------------------------------------:)

Parameter Adt ComparableElem2;
Interface
  Use  Booleans;
  Sort elem2;
  Operation
    _ = _ : elem2 elem2 -> boolean;
Body
  Theorems
  ;; usual equivalence relation properties
  ;; reflexivity
  (x = x) = true;
  ;; symmetry
  (x = y) = true => (y = x) = true;
  ;; transitivity
  (x = y) = true & (y = z) = true => (x = z) = true;
  Where
    x, y, z : elem2;
End ComparableElem2;
```

```
(:------------------------------------------------------------------------*
| Generic list specification.                                             |
|                                                                         |
| List type is defined over emptylist and cons generators. Miscalenaous  |
| operations are provided. Functions are total, if they are not then     |
| emptylist is given. head is partial.                                   |
|                                                                         |
| Author  : O. Biberstein G. Dimarzo                                      |
| Date    : 15 June 95                                                    |
*------------------------------------------------------------------------:)

Generic Adt List(ComparableElem);
Interface
  Use  Naturals, Booleans;
  Sort list;
  Generators
    []    : -> list;
    _ ' _ : elem  list -> list;
  Operations
    _ | _         : list list -> list;           ;; concatenation
    # _           : list -> natural;             ;; number of component
    take _ from _ : natural list -> list;        ;; first n component
    drop _ from _ : natural list -> list;        ;; l - take(n,l)
    head _        : list -> elem;                ;; first component
    tail _        : list -> list;                ;; l - first component
    empty? _      : list -> boolean;
    reverse _     : list -> list;
    _ = _         : list list -> boolean;
Body
  Axioms
    ([] | l1)  = l1;
    ((e ' l1) | l2) = (e ' (l1 | l2));
    #([]) = 0;
    #(e ' l1) = succ(#(l1));
    take n from []       = [];
    take 0 from e ' l1    = [];
    take succ(n) from e ' l1 = e ' take n from l1;
    drop n from []       = [];
    drop 0 from (e'l1)    = (e'l1);
    drop succ(n) from e ' l1 = (drop n from l1);
    head(e'l1) = e;
    tail([]) = [];     ;; if is-empty(l1) then tail(l1) = []
    tail(e'l1) = l1;
    empty?([]) = true;
    empty?(e'l1) = false;
    reverse([]) = [];
    reverse(e'l1) = (reverse l1) | (e'[]);
    ([] = []) = true;
    ( e'l1 = []) = false;
    ([] = e'l1) = false;
    (e1'l1 = e2'l2 ) = (e1 = e2) and (l1 = l2);
  Theorem
    (take n from l ) | (drop n from l) = l;
    reverse(reverse(l)) = l;
    ;; usual equivalence relation properties
    (l = l) = true;                           ;; reflexivity
    (l1 = l2) = true => (l2 = l1) = true;     ;; symmetry
    ;; transitivity
    (l1 = l2) = true & (l2 = l3) = true => (l1 = l3) = true;
  Where
    l, l1, l2, l3 : list;
    e, e1, e2     : elem;
    n : natural;
End List;
```

```
(:-----------------------------------------------------------------------*
| Cartesian product specification.                                       |
|                                                                        |
| Generic cartesian product pc2 type is defined over pair generator.     |
| Projection et equality operations are provided.                        |
|                                                                        |
| Revised : G. Di Marzo                                                  |
| Date    : 15 June 1995                                                 |
*-----------------------------------------------------------------------:)

Generic Adt Pc2(ComparableElem1, ComparableElem2);
Interface
  Use  Booleans;
  Sort pc2;
  Generator
    < _ _ >  : elem1 elem2 -> pc2;
  Operations
    fst _ : pc2 -> elem1;                       ;; first projection
    snd _ : pc2 -> elem2;                       ;; second projection
    _ = _ : pc2 pc2 -> boolean;
Body
  Axioms
    fst(<x y>) = x;
    snd(<x y>) = y;
    ( <x y> = <u v>) = (x = u) and (y = v);
  Where
    x, u : elem1;
    y, v : elem2;
End Pc2;
```

```
(:---------------------------------------------------------------------*
| Lifo queue (stack)                                                   |
|                                                                      |
| Several operations are provided, for instance, pop, top etc..        |
|                                                                      |
| Revised : D. Buchs                                                   |
| Date    : 5 July 1995                                                |
*---------------------------------------------------------------------:)

Generic Adt Lifo(Elem);
Interface
  Use  Naturals, Booleans;
  Sort lifo;
  Generators
    []    : -> lifo;
    _ ' _ : lifo elem -> lifo;
  Operations
    push _ _ : lifo elem -> lifo;
    empty? _ : lifo -> boolean;
    pop _    : lifo -> lifo;
    top _    : lifo -> elem;
    # _      : lifo -> natural;
Body
  Axioms
    push lf e = lf'e;

    empty? [] = true;
    empty? lf'e = false;

    pop [] = [];
    pop (lf'e) = lf;

    top lf'e = e;

    # [] = 0;
    # lf'e = succ #lf;
  Where
    lf : lifo;
    e  : elem;
End Lifo;
```

```
(:-----------------------------------------------------------------------*
| Fifo queue                                                             |
|                                                                        |
| Several operations are provided for instance insert, remove, next ... |
|                                                                        |
| Revised : D. Buchs                                                     |
| Date    : 5 July 1995                                                  |
*-----------------------------------------------------------------------:)

Generic Adt Fifo(ComparableElem);
Interface
  Use  Naturals, Booleans;
  Sort fifo;
  Generators
    []    : -> fifo;
    _ ′ _ : fifo elem  -> fifo;
  Operations
    insert _ _ : fifo elem  -> fifo;
    _ | _      : fifo fifo -> fifo;
    empty? _   : fifo -> boolean;
    remove _   : fifo -> fifo;
    next _     : fifo -> elem ;
    _ = _      : fifo fifo -> boolean;
    # _        : fifo -> natural;
   _ inside? _ : elem fifo -> boolean;
Body
  Axioms
    insert ff e = ff ′ e;
    empty? [] = true;
    empty? ( ff ′ e) = false;
    remove [] = [];
    remove ([] ′ e) = [];
    empty? ff = false => remove (ff ′ e) = (remove ff) ′ e;
    next ([] ′ e) = e;
    empty? ff = false => next (ff ′ e) = next ff;

    e inside? [] = false;
    e inside? ff ′ ee = (e = ee) or (e inside? ff);

    # [] = 0;
    # ff ′ e = succ(#ff);
  Where
    ff    : fifo;
    e, ee : elem;
End Fifo;
```

```
(:-----------------------------------------------------------------------*
| Set specification (with axioms between generators)                     |
|                                                                        |
| Revised : D. Buchs                                                     |
| Date    : 5 July 1995                                                  |
*-----------------------------------------------------------------------:)

Generic Adt Set(ComparableElem);
Interface
  Use Naturals, Booleans;
  Sorts set;
  Generators
    []    : -> set;
    _ ’ _ : set elem -> set;
  Operations
    _ + _       ,
    _ - _       : set set  -> set;
    _ inside? _ : elem set -> boolean;
    _ - _       : set elem -> set;
    _ = _       ,
    _ <= _      ,
    _ < _       ,
    _ > _       ,
    _ >= _      : set set -> boolean;
    # _         : set -> natural;
Body
  Axioms
  ;; commutativity of the insertion
  ((s  ’ e1) ’ e2) = ((s ’ e2) ’ e1);
  ;; no duplication in set
  ((s’ e1)’ e1) = (s ’ e1) ;
  [] + s  = s;
  (s1 ’ e1) + s2 = (s1 + s2) ’ e1;
  s - []  = s;
  s - (s1 ’ e1)  = (s - e1) - s1;
  (e1 = e2) = true => ((s ’ e1) -  e2) = s;
  (e1 = e2) = false => ((s ’ e1) -  e2) = (s - e2) ’ e1;
  # [] = 0;
  # (s ’ e1) = succ(# s);
  (s1 = s2) = (s1 <= s2) and (s2 <= s1);
  e1 inside? [] = false;
  (e1 inside? (s ’ e2)) = (e1 = e2) or (e1 inside? s) ;
  [] <= [] = false;
  (s ’ e1) <= [] = false;
  [] <= (s ’ e1) = true;
  (e1 = e2) = true => (s1’e1) <= (s2’e2) = s1 <= s2;
  (e1 = e2) = false => ((s1’e1) <= (s2’e2)) = (e1 inside? s2) and s1 <= (s2 ’ e2);

  s1 < s2  = s1 <= s2 and not(s2 <= s1);
  s1 >= s2 = s2 <= s1;
  s1 > s2  = s2 < s1;
  Where
    e1, e2    : elem;
    s, s1, s2 : set;
End Set;
```

```
(:-----------------------------------------------------------------------*
| Bags (sets with repetition)                                            |
|                                                                        |
| Revised : D. Buchs                                                     |
| Date    : 5 July 1995                                                  |
*-----------------------------------------------------------------------:)

Generic Adt Bag(ComparableElem);
Interface
  Use  Naturals, Booleans;
  Sort bag ;
  Generators
    []    : -> bag ;
    _ ' _ : bag elem -> bag ;
  Operations
    _ + _        ,
    _ - _        : bag bag  -> bag;
    _ inside? _  : elem bag -> boolean;
    _ - _        : bag elem -> bag;
    _ = _        ,
    _ <= _       ,
    _ < _        ,
    _ > _        ,
    _ >= _       : bag bag -> boolean;
    # _          : bag -> natural;
Body
  Axioms
    ;; commutativity of the insertion
    ((s'e1)'e2) = ((s'e2)'e1);

    [] + s  = s;
    (s1'e1) + s2 = (s1 + s2)'e1;

    s - []  = s;
    s - (s1'e1)  = (s - e1) - s1;

    (e1 = e2) = true => ((s'e1) -  e2) = s;
    (e1 = e2) = false => ((s'e1) -  e2) = (s - e2)'e1;

    # [] = 0;
    # (s'e1) = succ(# s);

    (s1 = s2) = (s1 <= s2) and (s2 <= s1);

    e1 inside? [] = false;
    (e1 inside? (s'e2)) = (e1 = e2) or (e1 inside? s) ;

    [] <= [] = false;
    (s'e1) <= [] = false;
    [] <= (s'e1) = true;
    (e1 = e2) = true => (s1'e1) <= (s2'e2) = s1 <= s2;
    (e1 = e2) = false => ((s1'e1) <= (s2'e2)) = (e1 inside? s2) and s1 <= (s2'e2);
    s1 < s2 = s1 <= s2 and not(s2 <= s1);
    s1 >= s2 = s2 <= s1;
    s1 > s2 = s2 < s1;
  Where
    e1, e2   : elem;
    s,s1, s2 : bag;
End Bag;
```

```
(:-------------------------------------------------------------------------*
| Table data structures                                                    |
|                                                                          |
| ! the commutation and duplication properties (axioms #1 and #2) are      |
| operationnaly critical (to be change in an observationnal spec)          |
|                                                                          |
| Revised : D. Buchs                                                       |
| Date    : 5 July 1995                                                    |
*-------------------------------------------------------------------------:)

Generic Adt Table(ComparableElem);
Interface
  Use Naturals, Booleans;
  Sorts table;
  Generators
    []          : -> table;
    _ [ _ ] = _ : table natural elem -> table;
  Operations
    _ | _   : table table -> table;
    _ [ _ ] : table natural -> elem;
    shift _ : table -> table;
    # _     : table -> natural;
    _ = _   ,
    _ <= _  : table table -> boolean;
    _ at _ inside? _ : elem natural table -> boolean;
Body
  Axioms
    ;; commutativity of the insertion
    (pos1 = pos2) = false =>
      ((t [ pos1 ] = e1)[ pos2 ] = e2) = (((t [ pos2 ]  = e2) [ pos1 ]) = e1);
    ;; no duplication in table
    (pos1 = pos2) = true =>
      ((t [ pos1 ] = e1)[ pos2 ] = e2) = ((t [ pos2 ]) = e2);
    [] | t = t;
    (t1 [ pos1 ] = e1) | t2 = (t1 | t2) [ pos1 ] = e1;
    shift(t [ pos1 ] = e1) = (shift t) [ pos1 ] = e1;
    shift([]) = [];
    (pos1 = pos2) = false => ((t [ pos1 ] = e1)[ pos2 ]) = (t [ pos2 ]);
    (pos1 = pos2) = true => ((t [ pos1 ] = e1)[ pos2 ]) = e1 ;
    # [] = 0;
    # (t [ pos1 ] = e1) = succ(# t);
    (t1 = t2) = (t1 <= t2) and (t2 <= t1);
    e1 at pos1 inside? [] = false;
    (pos1 = pos2) = true => (e1 at pos1 inside? (t [ pos2 ] = e2)) = (e1 = e2);
    (pos1 = pos2) = false =>
      (e1 at pos1 inside? (t [ pos2 ] = e2)) = (e1 at pos1 inside? t);
    [] <= [] = false;
    (t [ pos1 ] = e1) <= [] = false;
    [] <= (t [ pos1 ] = e1) = true;
    (pos1 = pos2) = true =>
      ((t1 [ pos1 ] = e1) <= (t2 [ pos2 ] = e2)) = (e1 = e2) and (t1 <= t2);
    (pos1 = pos2) = false =>
      ((t1 [ pos1 ] = e1) <= (t2 [ pos2 ] = e2)) =
      (e1 at pos1 inside? t2) and (t1 <= (t2 [ pos2 ] = e2));
  Where
    pos1, pos2 : natural;
    t, t1, t2  : table;
    e1, e2     : elem;
End Table;
```

```
(:-----------------------------------------------------------------------*
| Generic list specification (with lexical order).                       |
|                                                                        |
| List type is defined over emptylist and cons generators. Miscalenaous |
| operations are provided. Functions are total, if they are not then     |
| emptylist is given. head is partial.                                   |
| Lexical order is provided in addition to the usual List module.       |
|                                                                        |
| Revised : D. Buchs                                                     |
| Date    : 5 july 1995                                                  |
*-----------------------------------------------------------------------:)

Generic Adt OrderedList(OrderedElem);
Interface
  Use  Booleans,Naturals;
  Sort orderedlist;
  Generators
    []    : -> orderedlist;
    _ ' _ : elem orderedlist -> orderedlist;
  Operations
    _ | _          : orderedlist orderedlist -> orderedlist; ;; concatenation
    # _            : orderedlist -> natural;                 ;; number of components
    take _ from _ : natural orderedlist -> orderedlist;      ;; first n component
    drop _ from _ : natural orderedlist -> orderedlist;      ;; l - take(n,l)
    head _         : orderedlist -> elem;                    ;; first component
    tail _         : orderedlist -> orderedlist;             ;; l - first component
    empty? _       : orderedlist -> boolean;
    reverse _      : orderedlist -> orderedlist;
    _ = _          ,
    _ < _          ,
    _ <= _         ,
    _ > _          ,
    _ >= _         : orderedlist orderedlist -> boolean;
Body
  Axioms
    ([] | l1)  = l1;
    ((e'l1) | l2) = (e'(l1 | l2));

    #([]) = 0;
    #(e'l1) = succ(#(l1));

    take n from []      = [];
    take 0 from e'l1    = [];
    take succ(n) from e'l1 = e'take n from l1;

    drop n from []      = [];
    drop 0 from (e'l1)    = (e'l1);
    drop succ(n) from e'l1 = (drop n from l1);

    head(e'l1) = e;

    ;; if is-empty(l1) then tail(l1) = []
    tail([]) = [];
    tail(e'l1) = l1;

    empty?([]) = true;
    empty?(e'l1) = false;

    reverse([]) = [];
    reverse(e'l1) = (reverse l1) | (e'[]);

    ([] = []) = true;
    ( e'l1 = []) = false;
    ([] = e'l1) = false;
    (e1'l1 = e2'l2 ) = (e1 = e2) and (l1 = l2);

    ([] < [])     = false;
    ([] < (e'x )) = true;
    ((e'x ) < []) = false;
    (e1 < e2) = true => ((e1'x ) < (e2'y )) = true;
    (e1 = e2) = true => ((e1'x ) < (e2'y )) = x < y;
    (e2 < e1) = true => ((e1'x ) < (e2'y )) = false;

    (x <= y) = ((x = y) or (x < y));

    (x > y) = (y <= x);

    (x >= y) = (y < x);
  Theorem
    (take n from l ) | (drop n from l) = l;
    reverse(reverse(l)) = l;
    ;; usual equivalence relation properties
    ;; reflexivity
    (l = l) = true;
    ;; symmetry
    (l1 = l2) = true => (l2 = l1) = true;
    ;; transitivity
    (l1 = l2) = true & (l2 = l3) = true => (l1 = l3) = true;
  Where
    l, l1, l2, l3 : orderedlist;
    e, e1, e2     : elem;
    x, y          : orderedlist;
    n             : natural;
End OrderedList;
```

```
;; Standard Library of ADTs and Objects
;;
;; constructortypes.sys

Specification   constructortypes;
Version         1.0;
Date            12 September 1995;

Authors D.Buchs :
        Item, ComparableItem, ComparableItem1,ComparableItem2, OrderedItem,
        List, ListwithOrder, Pc2, Fifo, Lifo, Set,
        Bag, Table;

Usebasictypes.sys;

Modules Item, ComparableItem, ComparableItem1, ComparableItem2, OrderedItem,
        List, OrderedList, Pc2, Fifo, Lifo, Set,
        Bag, Table : constructortypes;

End constructortypes;
```

```
(:-------------------------------------------------------------------------*
| Specifications of composed modules  'structuredtypes'.                 |
|                                                                         |
| Semantics: The semantics used is based on the implicit definition of   |
| the validity domain of the functions by the axioms. The possible       |
| values are determined by the generators (finitely generated with       |
| respect to the generators). A function is not defined if it has no     |
| axiom for particular value.                                             |
|                                                                         |
| Modules provided:                                                       |
|                                                                         |
|  Structured types:                                                      |
|    Coord, String.                                                       |
|                                                                         |
| Author  : D. Buchs previous versions from O.Biberstein and G.di Marzo |
| Date    : 5 July 1995                                                   |
| Revised : O. Biberstein                                                 |
| Date    : 95/10/20                                                      |
*-------------------------------------------------------------------------:)


(:-------------------------------------------------------------------------*
| Coordinate (instantiation of Pc2)                                       |
|                                                                         |
| Several operations are: Projection on x and y.                          |
|                                                                         |
| Revised: D. Buchs                                                       |
| Date   : 5 July 1995                                                    |
*-------------------------------------------------------------------------:)

Adt Coord As Pc2(Integers, Integers);
Morphism
  elem1 -> integer;
  elem2 -> integer;
  _ = _ In ComparableElem1  ->  _ = _ In Integers;
  _ = _ In ComparableElem2  ->  _ = _ In Integers;
Rename
  pc2   -> coord;
  fst _ -> x-coord _ ;
  snd _ -> y-coord _ ;
Interface
Body
End Coord;

(: A CHANGER, Olivier:)

(:-------------------------------------------------------------------------*
| Coordinate (instantiation of Pc2)                                       |
|                                                                         |
| Several operations are Projection on x and y.                           |
|                                                                         |
| Revised: D. Buchs                                                       |
| Date   : 5 July 1995                                                    |
*-------------------------------------------------------------------------:)

Adt String As OrderedList(Characters);

Morphism
  character -> elem;
  _ = _  ->  _ = _ ;
  _ < _  ->  _ < _ ;
Rename list -> string;
Interface
Body
End String;

;; Standard Library of ADT
;;
;; structuredtypes.sys

Specification   structuredtypes;
```

```
Version         1.0;
Date            5 July 1995;

Author  D.Buchs :  Coord, String;

Use     basictypes.sys, constructortypes.sys;

Modules Coord, String : structuredtypes.coopn;

End structuredtypes;
```

B Syntactic Aspects

Olivier Biberstein & Mathieu Buffo

A CO-OPN$_{1.5}$ specification is composed of many modules which may be distributed over several files. Thus, a description of the various files composing a sepecification as well as their content is required. Such a description is expressed by means of a *system file* which describes which files and which modules compose the specification.

Section B.1 describes the lexical elements of the CO-OPN$_{1.5}$ language and the system files. Section B.2 and B.3 show, respectively, the BNF-like grammar of the CO-OPN$_{1.5}$ language as well as of the system files.

B.1 Lexical Elements of the CO-OPN$_{1.5}$ Language

B.1.1 Character Set

A text written in CO-OPN$_{1.5}$ is composed of characters of the ISO Latin 1 character set (ISO 8859 1). In our context, this set is divided in four sub-sets :

Alphanum : All letters of ISO Latin 1 and the characters `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`,

Blank : the characters *Space*, *Tab* and *New Line*,

Separator : the characters '`.`', '`,`', '`:`', '`;`', '`_`', '`(`', '`)`', '`"`',

Special : all others printable characters of the ISO Latin 1 character set.

All others characters (in particular the non-printable ones) generate a lexical error when non encountered in comments.

B.1.2 Reserved Symbols

All the reserved symbols are listed below :

'`::`', '`->`', '`=>`', '`..`', '`//`', '`(:`', '`:)`', '`;;`'.

The **Reserved** sub-set denotes the reserved symbols which are involved in the tokens delimitation. This sub-set is composed of the following symbols : '`::`', '`->`', '`=>`', '`..`', '`//`'.

B.1.3 Tokens

The lexical tokens are either delimited by the elements of the **Reserved** sub-set or by the characters which are not in the **Alphanum** sub-set. The characters of the **Blank** sub-set are skipped.

B.1.4 Reserved Words

The reserved words are English words. They are neither case-sensitive nor singular/plural-sensitive. These words are :

> **`As`**, **`Axiom`**, **`Body`**, **`Adt`**, **`End`**, **`Generator`**, **`Generic`**, **`In`**, **`Initial`**, **`Interface`**, **`Method`**, **`Module`**, **`Morphism`**, **`Object`**, **`Operation`**, **`Parameter`**, **`Place`**, **`Rename`**, **`Sort`**, **`Theorem`**, **`Transition`**, **`Use`**, **`Where`**, **`With`**,

B.1.5 Word

A word is either a unique character of the **Special** sub-set or a sequence of characters of the **Alphanum** sub-set.

Examples :

1. are valid words :
   `transmit`, `Send`, `2BeOrNot2Be`, `9`, `28`, `$`, `n'`, `n''`.
2. are not valid words :
   `more or less`, `n"`, `obj.get`, `is_empty`, `spring->summer`, **`Body`**.

B.1.6 Comments

Two types of comments are allowed.

- The *multi-line* comments start with the symbol '`(:`' and end with the symbol '`:)`'. These comments can be nested.
- The *single-line* comments start with the symbol '`;;`' and end with the end of the line.

Examples :

1.
```
(: This object is an unbounded buffer
 : which contains natural numbers
 :)
Object Buffer;
Interface
  Use Nat;
  Methods
    put _ : nat,   ;; add a natural number
    get;           ;; remove a natural number
Body               ;; the implementation is not yet provided
End Buffer;
```
2.
```
(: This is a nested (: multi-line
     comment :) :)
```

B.2 BNF-like Grammar of the CO-OPN$_{1.5}$ Language

The BNF-like grammars given below have adopted the following conventions :

- bold face names denote terminal keywords,
- plain names denote non terminal keywords,
- quoted symbols denote terminals,
- $(\alpha)^*$ means zero or many α,
- $(\alpha)^+$ means one or more α,
- $[\alpha]$ means optional α,

where α is any sequence of terminal or non terminal.

The "Word" non terminal which is not present in the grammar corresponds to the element described in section B.1.5.

Main ⟶ (Module)$^+$

Module ⟶
[(**Generic** | **Parameter**)] **Adt** [**Module**] ModuleId
Header
AlgInterface
AlgBody
End ([ModuleId '`;`'] | ['`;`']) |
[(**Generic** | **Parameter**)] **Object** [**Module**] ModuleId
Header
ObjInterface
ObjBody
End ([ModuleId '`;`'] | ['`;`'])

Header ⟶
[ModuleParam] '`;`'
[**Morphism** MorphismBloc]
[**Rename** RenameBloc]
|
[ModuleParam] **As** ModuleId [ModuleParam] '`;`'
[**Morphism** MorphismBloc]
[**Rename** RenameBloc]

ModuleParam ⟶ '`(`' ModuleId ('`,`' ModuleId)* '`)`'

MorphismBloc ⟶ (MorphismList '`;`')$^+$

MorphismList ⟶ MorphismTerm | MorphismTerm ('`,`' MorphismTerm)*

MorphismTerm ⟶ MixIdentifier [**In** ModuleId] '`->`' MixIdentifier [**In** ModuleId]

RenameBloc ⟶ (RenameList '`;`')$^+$

RenameList ⟶ RenameTerm | RenameTerm (‘,’ RenameTerm)*

RenameTerm ⟶ MixIdentifier [**In** ModuleId] ‘->’ MixIdentifier

AlgInterface ⟶
  Interface
  [**Use** UseBloc]
  [**Sort** SortBloc]
  [**Generator** GenOperBloc]
  [**Operation** GenOperBloc]

AlgBody ⟶
  Body
  [**Use** UseBloc]
  [**Sort** SortBloc]
  [**Generator** GenOperBloc]
  [**Operation** GenOperBloc]
  [**Axiom** AlgFormulaBloc]
  [**Theorem** AlgFormulaBloc]
  [**Where** VariableBloc]

ObjInterface ⟶
  Interface
  [**Use** UseBloc]
  [**Method**MethodBloc]

ObjBody ⟶
  Body [**Use** UseBloc]
  [**Method** MethodBloc]
  [**Transition** TransitionBloc]
  [**Place** PlaceBloc]
  [**Initial** InitialBloc]
  [**Axiom** ObjFomulaBloc]
  [**Theorem** ObjFormulaBloc]
  [**Where** VariableBloc]

UseBloc ⟶ (UseList ‘;’)$^+$

UseList ⟶ ModuleId (‘,’ ModuleId)*

SortBloc ⟶ (SortList ‘;’)$^+$

SortList ⟶ SortId (‘,’ SortId)*

GenOperBloc ⟶ (GenOperList ‘:’ [[Type] ‘->’] Type ‘;’)$^+$

GenOperList ⟶ GenOperId (‘,’ GenOperId)*

AlgFormulaBloc $\longrightarrow$ (AlgFormulaTerm '`;`')$^+$

AlgFormulaTerm $\longrightarrow$ [AxiomId '`::`'] [Condition '`=>`'] Term

MethodBloc $\longrightarrow$ (MethodList ['`:`' Type] '`;`')$^+$

MethodList $\longrightarrow$ MethodId ('`,`' MethodId)*

TransitionBloc $\longrightarrow$ (TransitionList '`;`')$^+$

TransitionList $\longrightarrow$ TransitionId ('`,`' TransitionId)*

PlaceBloc $\longrightarrow$ (PlaceList '`:`' Type ['`(`' Identifier '`)`'] '`;`')$^+$

PlaceList $\longrightarrow$ PlaceId ('`,`' PlaceId)*

InitialBloc $\longrightarrow$ (Marking '`;`')$^+$

ObjFormulaBloc $\longrightarrow$ (ObjFormulaTerm '`;`')$^+$

ObjFormulaTerm $\longrightarrow$
[AxiomId '`::`'] [Condition '`=>`'] Event [**`With`** Synchronization] '`:`'
Marking '`->`' Marking

Condition $\longrightarrow$ Term

Event $\longrightarrow$ Term

Synchronization $\longrightarrow$ Term

Marking $\longrightarrow$ Term

Synchronization $\longrightarrow$ Term

Marking $\longrightarrow$ Term

Type $\longrightarrow$ TermNoPar

VariableBloc $\longrightarrow$ (VariableList '`:`' Type '`;`')$^+$

VariableList $\longrightarrow$ VariableId ('`,`' VariableId)*

ModuleId $\longrightarrow$ Word

AxiomId $\longrightarrow$ Word

TransitionId $\longrightarrow$ Identifier

SortId $\longrightarrow$ Identifier

VariableId $\longrightarrow$ Identifier

TransitionId $\longrightarrow$ Identifier

MethodId $\longrightarrow$ MixIdentifier

PlaceId $\longrightarrow$ MixIdentifier

GenOperId $\longrightarrow$ MixIdentifier

Term $\longrightarrow$ TermPar (‘`,`’ TermPar)*

TermPar $\longrightarrow$ TermNoPar | (‘`(`’ Term ‘`)`’ [TermNoPar])$^+$

TermNoPar $\longrightarrow$ TermNoParFactor (‘`,`’ TermNoParFactor)*

TermNoParFactor $\longrightarrow$ (Word [**In** ModuleId])$^+$

MixIdentifier $\longrightarrow$ ‘_’ | Identifier | ‘_’ Identifier | MixIdentifier ‘_’ | MixIdentifier ‘_’ Identifier

Identifier $\longrightarrow$ (Word)$^+$

B.3 Syntax of the System Files

A system file describes which modules compose a specification and where are located (in which file) these modules. The lexical aspects of the systems files are the same as the ones of section B.1 except that the *reserved words* are the followings :

> **`Author`**, **`Date`**, **`End`**, **`Module`**, **`Use`**, **`Version`**, **`Specification`**.

The BNF-like grammar of the system files is as follows :

Main $\longrightarrow$
  `Specification` Identifier ‘`;`’
  [**`Version`** ExtIdentifier ‘`;`’]
  [**`Date`** ExtIdentifier ‘`;`’]
  [**`Author`** AuthorBloc]
  [**`Use`** UseBloc]
  [**`Module`** ModuleBloc]
  `End` ([Identifier ‘`;`’] | [‘`;`’])

AuthorBloc $\longrightarrow$ (AuthorList [‘`:`’ ExtIdentifier] ‘`;`’)$^+$

AuthorList $\longrightarrow$ AuthorId (‘`,`’ AuthorId)*

UseBloc $\longrightarrow$ (UseList ‘`;`’)$^+$

UseList $\longrightarrow$ FileId (‘`,`’ FileId)*

ModuleBloc $\longrightarrow$ (ModuleList ‘`:`’ FileName ‘`;`’)$^+$

ModuleList $\longrightarrow$ ModuleId (‘`,`’ ModuleId)

ModuleId $\longrightarrow$ Word

AuthorId $\longrightarrow$ Identifier

FileId $\longrightarrow$ Identifier

ExtIdentifier $\longrightarrow$ (Word | ‘`.`’ | ‘`,`’)$^+$

Identifier $\longrightarrow$ (Word | ‘`.`’)$^+$

Examples :

The system file given in Figure B.1 and B.2 show, respectively, the system files of the Alternate Bit Protocol example and the lift example introduced in sections 3.2 and 3.3 (see pages 15 and 16).

```
;; Alternate Bit Protocol
;;
;; abp.sys

Specification   abp;
Version         1.0;
Date            23 oct 1995;

Authors O.Biberstein, G. Di Marzo :
        Message, AltBit, Fifo, Frame,
        Receiver, Transmitter, EtherIn, EtherOut;

Use     basictypes.sys;

Modules Message, Altbit, Fifo, Frame : abpadt.coopn;
        Receiver, Transmitter, EtherIn, EtherOut : abpobj.coopn;

End abp;
```

Figure B.1: The System File of the Alternate Bit Protocol Example

```
;; The Lift
;;
;; lift.sys

Specification lift;
Version       1.0;
Date          23 oct 1995;

Authors O.Biberstein, G. Di Marzo :
        Direction, Goal, Floors, ListGoals,
        Cabin, BuildingFloors, Control;

Use     basictypes.sys, constructortypes.sys;

Modules Direction, Goal, Floors, ListGoals,
        Cabin, BuildingFloors, Control : lift.coopn;

End lift;
```

Figure B.2: The System File of the Lift Example

Bibliography

[BB94] Stéphane Barbey and Didier Buchs. Testing Ada Abstract Data Types using Formal Specification. Tech. Report 94, Ecole Polytechnique Fédérale de Lausanne, 1994.

[BB95] Olivier Biberstein and Didier Buchs. Structured Algebraic Nets with Object-Orientation. In *Proceedings of the "Application and Theory of Petri Nets 1995" workshop on "Object-Oriented Programming and Models of Concurrency"*, Torino, Italy, June 1995.

[BFR93a] Didier Buchs, Jacques Flumet, and Pascal Racloz. SANDS: Structured algebraic net development system. In Buy Ugo, editor, *14th International Conference on Application and Theory of Petri Nets, Tool presentation abstracts*, pages 25–29. Chicago, USA, June 1993.

[BFR93b] Didier Buchs, Jacques Flumet, and Pascal Racloz. Sands: Structured algebraic net development system. Cahiers du CUI 71, University of Geneva, 1993.

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

[BURB94] Mathieu Buffo, Erik Urland, José Rolim, and Didier Buchs. Le projet TSPP. Cahiers du CUI 91, Centre Universitaire Informatique, CUI, Université de Genève, 24, rue du Général Dufour, 1211 Genève 4, Suisse, December 1994.

[CK90] Christine Choppy and Stéphane Kaplan. Mixing abstract and concrete modules: Specification, development and prototyping. In *12th International Conference on Software Engineering*, pages 173–184, Nice, March 1990.

[Flu95] Jacques Flumet. *Un environnement de développement de spécifications pour systèmes concurrents.* PhD thesis, Université de Genève, 1995. to appear.

[GBD$^+$93] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM3 user's guide and reference manual. Technical Manual 12187, Oak Ridge National Laboratory, 1993.

[RB93] Pascal Racloz and Didier Buchs. Symbolic proof of CTL formulae over Petri nets. In G. Levent, O. Raif, and G. Erol, editors, *8th Internationnal Symposium on Computer and Information Sciences*, pages 189–196. Istambul, Turkey, November 1993.

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

[Tan89] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, second edition, 1989.