A PROPOSAL FOR A GRAPHIC SYNTAX FOR LOTOS

draft

T. Bolognesi *
D. Latella *
A. Pisano **

Rapporto Interno C88-01

* C.N.R.- CNUCE, Via S. Maria 36 - 56100 - Pisa - ITALY
** TECSIEL, Via S. Maria 19 - 56100 - Pisa - ITALY

Consiglio Nazionale delle Ricerche
Istituto CNUCE - Pisa
Gennaio 1988

A PROPOSAL FOR A GRAPHIC SYNTAX FOR LOTOS

Tommaso Bolognesi *, Diego Latella *, Annamaria Pisano **

* C.N.R.- CNUCE, Via S. Maria 36 - 56100 - Pisa
** TECSIEL, Via S. Maria 19 - 56100 - Pisa

A graphic syntax for the ISO formal description technique LOTOS is introduced. The syntax is meant to improve the readability of LOTOS behaviour expressions, and to highlight aspects such as sequentiality, parallelism, synchronization and choice, without excessively departing from the structure of the traditional 'textual' syntax. A formal scheme based on Prolog is adopted to derive pictures from the abstract syntax of the language. One complete example of a graphic specification is given.

0. INTRODUCTION

The advantages in adopting Formal Description Techniques (FDT's) for the specification of communication systems have been widely recognized within the community of system designers and developers, and also within the standardization institutions of ISO and CCITT. FDT's allow one to develop unambiguous specifications which serve as a sound basis for the analysis and the implementation of a system. However, a disadvantage of FDT's is that people are required to learn and understand them. Thus, in order to achieve a wide-spread acceptance of FDT's, they must be easy to learn and easy to use. While the suitability of a specification language to express the relevant features of a system depends essentially on the *formal semantics* of that language, it is a fact that the success and 'popularity' of the latter may be substantially influenced by the adopted 'syntactic sugar'.

The importance of a concrete graphic syntax for an FDT was first recognized within CCITT. In fact, the original form of SDL (Specification and Description Language, [CCITT87]) was SDL / GR (Graphic Representation), based on a set of standardized graphic symbols.

LOTOS ([ISO-8807]) is expected to reach the status of International Standard during 1988. Furthermore CCITT has recognized the applicability of LOTOS for the specification of communication systems, and, based on the experience with SDL / GR, CCITT experts have suggested to start a cooperation with ISO for defining an alternative, graphical representation for LOTOS. During the year 1987 an unespectedly large number of preliminary proposals for a graphical LOTOS have circulated among CCITT and ISO experts, and have been discussed during a joint meeting between the two organization (see, for instance, [T87], [BLMP87], [B87], [BT87], [NQ87], [AS87]). In the meanwhile, a "new Question" has been proposed within ISO to handle this topic [ISO-Q48.4], with the purpose to produce an addendum to the LOTOS standard, and a satisfactory support to such an activity has been expressed by ISO Member Bodies.

Although our current proposal is somewhat different from the previous proposals mentioned above, we think we have taken into account the indication implicit in some of them, which supports the *simplicity* of graphic LOTOS. In particular, as a novel feature, we have conceived the work space of a LOTOS user as an imaginary desk-top where paper sheets of rounded or rectangular shapes may be distributed with sufficient freedom, and may partially overlap. Surprisingly enough, LOTOS gates can *speak* their offers to each other...

The paper is organized as follows. Section 1 lists a number of general principles for defining a graphic version for a FDT, and some specific criteria adopted in our proposal. The graphic representations of the syntactic constructs of LOTOS are informally introduced in Section 2, while Section 3 presents a complete example. The formalization of our proposal is contained in the three Appendices.

1. CRITERIA FOR DEFINING A GRAPHICAL LOTOS

It seems desirable to define graphic forms for LOTOS (called GLOTOS hereafter) with the following principles in mind.

Syntax-oriented. GLOTOS is primarily meant as an alternative concrete syntax for LOTOS. It should directly represent *all and only* the information contained in a LOTOS (textual) specification, in a way which highlights the structure of the latter (sequentiality, parallelism, synchronization...).

Improving readability. An appropriate use of indentation greatly favours the readability of LOTOS; GLOTOS potentially offers a less constrained use of the dimensions of a paper sheet, and may thus further improve readability, although it is not obvious that the improvement will be dramatic.

Easy two-way translation between LOTOS and GLOTOS. This translation should appear immediately to the reader of either form. In general, the translation of a complex construct must be obtained by combining the translations of its components.

Integrability of LOTOS and GLOTOS in the same specification. It should be possible to combine graphic and textual forms in the same specification. For instance, the designer may find convenient to write a simply structured process in LOTOS, within a specification in GLOTOS.

The LOTOS and GLOTOS syntaxes should be both derivable by the same abstract syntax. It is widely recognized that the abstract syntax is a central element for the formal definition of a language, and of its representations/interpretations. At the same time, the abstract syntax is essential for defining mixed syntax-directed editors for the language.

Easy to draw by hand. It should be easy to draw GLOTOS specifications by hand, on paper. Although it is likely that many specifications will be directly produced on a computer screen via CAD tools, the ability to do this by hand is seen as a guarantee that GLOTOS is kept simple.

Minimality. Among the different graphic forms which satisfactorily highlight the structure and features of a specification, preference should be given to those which minimize the complexity of the figures (number of lines, segments, symbols, icons...) on the paper sheet. A GLOTOS specification should not be substantially longer than its LOTOS version.

Full exploitation of the dimensions of a paper sheet. While the page of a LOTOS specification imposes, essentially, a uni-dimensional, top-down reading, GLOTOS should expand in both dimensions. It seems possible to exploit even the *third* dimension (z-axis), by partially overlapping symbols, or drawing some of them in perspective.

Flexibility to "natural" evolution. It is fair to recognize that an optimal definition of GLOTOS cannot be achieved in a single "burst of inspiration", but requires also a subsequent, evolutionary phase during which several allowed variants, or styles, are explored and compared, and some of them emerge over the others. It seems therefore desirable to leave some room, in the early definition of GLOTOS, for such an evolution, and to freeze some aspects of it only later (if at all).

The graphic syntax proposed here has been inspired by the principles listed in Section 1, and we hope that it satisfies them to an acceptable degree. More specific choices which characterize our proposal are given below.

Round shapes only for event-related elements. Circles and ovals are used to surround gates in process definitions, instantiations, synchronizations. An empty circle denotes full synchronization. Hidden gates are also surrouded by shapes with rounded edges. Value offers are included in cloud-like shapes with round edges, connected to circled gates. Round shapes are immediately identified in a GLOTOS complex figure.

Use of arrows. We use arrows *only* to indicate *semantic* sequentiality, that is sequentiality of events, never to indicate *syntactic* sequentiality, that is the sequentiality assumed of the *reading process*. The latter is often represented by *partial overlapping* of symbols (see next item).

Three-dimensionality. We exploit the third dimension of the paper sheet (z-axis) by partially overlapping symbols and by suggesting perspective (only in two cases). The partial overlapping of two or more symbols suggests that reading should proceed from the upper to the lower symbols in the ideal paper stack.

Preferred directions. Semantic and syntactic sequentiality imply movements left to right, or upper to lower, or upper-left to lower-right. Some freedom is allowed here, which can be used to optimize the layout of the graphic elements on the page.

2. INFORMAL INTRODUCTION TO THE GRAPHIC SYNTAX

We introduce a set of graphic representations for the syntactic constructs of LOTOS. Some familiarity with the syntax of LOTOS is assumed. The graphic forms are illustrated here in the quickest, completely informal way, by giving the graphic representation of each construct of the language, or of small combinations of them. Some graphic variants are also suggested. One complete example is given in Section 3. A formal definition of the proposed concrete graphic syntax of LOTOS, including the variants, is found in Appendix C. By convention, the "..." symbol and the *italics* text found in the following figures do not belong to the graphic syntax.

2.1 Observable and unobservable action prefix

Let us consider a process which offers to accept any value x of sort t, and to provide the value of expressions 'y + 1' and 'append(z, zqueue)' at gate 'beta1', provided that the guard '[x IsIn xset] is verified; after this, it may synchronize with other processes at gate 'beta2', without exchanging values; then it performs an internal, unobservable action, and then something else. Such a behaviour is described in LOTOS by the action prefix construct as follows:

```
beta1   ? x : nat, ! y+1 , ! append(z, zqueue) [x IsIn xset];
beta2;
i;
...
```

The process instantiation 'Data_Phase [g1, g2, g3, g4] (E_1, E_2, E_3)', which instantiates the behaviour of a process called 'Data_Phase' with the actual gates g1, ..., g4, and the actual parameters E_1, E_2 and E_3 (again these are LOTOS value expressions), is shown in Figure 2.3.

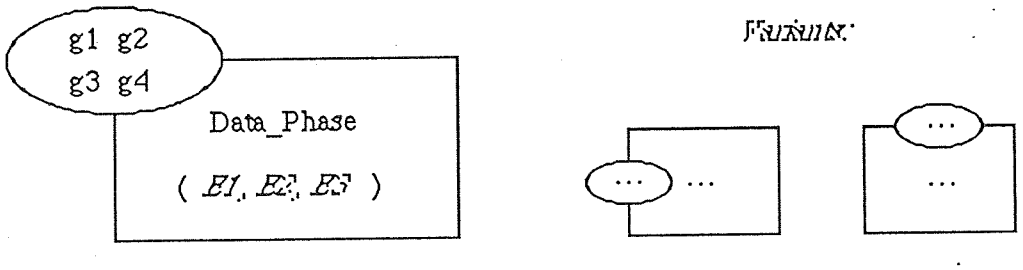


Figure 2.3

When a process has no externally visible gates, the oval shall be empty.

2.4 Process definition

The process definition

```
process Data_Phase [a1, a2, a3, b] (x, y : c_sort  h, k : d_sort) : exit (sort1, sort2) :=
        behaviour
where
...
endproc
```

is shown in Figure 2.4.

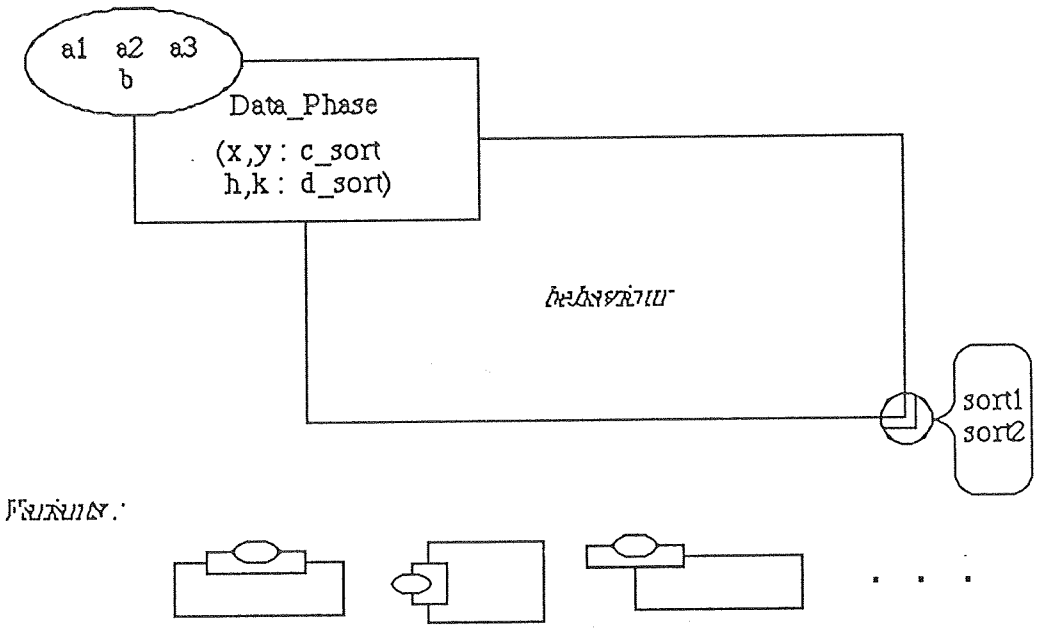


Figure 2.4

Note that a process definition looks like a process instantiation, except for a further rectangle containing the graphic description of the process behaviour (denoted by the word *behaviour* in figure). At the lower right corner of such a rectangle the indication is found of the process functionality 'exit (sort1, sort2)'. The functionalities 'exit' (without sort list) and 'noexit' are represented by using the same symbols used for the 'exit' and 'stop' processes (see Section 2.2). Note also that currently we do not provide a graphic representation of the 'where' clause found in a process definition. Similarly to the case of process instantiation, the oval in the definition of a process with no visible gates is empty.

2.5 Type definition and "library" construct

Currently we do not envisage any graphic representation for type definitions. Therefore we simply define two frames to surround the textual representations of these constructs; they are shown in Figure 2.5.

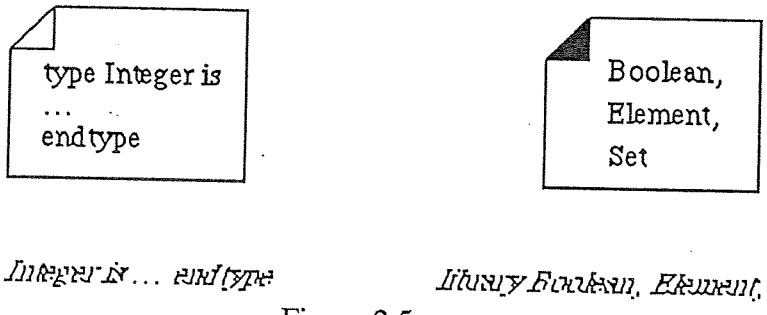


Figure 2.5

2.6 Hiding

The construct 'hide g1, g2 in *behaviour*' is shown in Figure 2.6.a. A nice shorthand (suggested by G. Scollo) for the unfrequent case when hiding is directly applied to a process instantiation is shown in Figure 2.6.b.

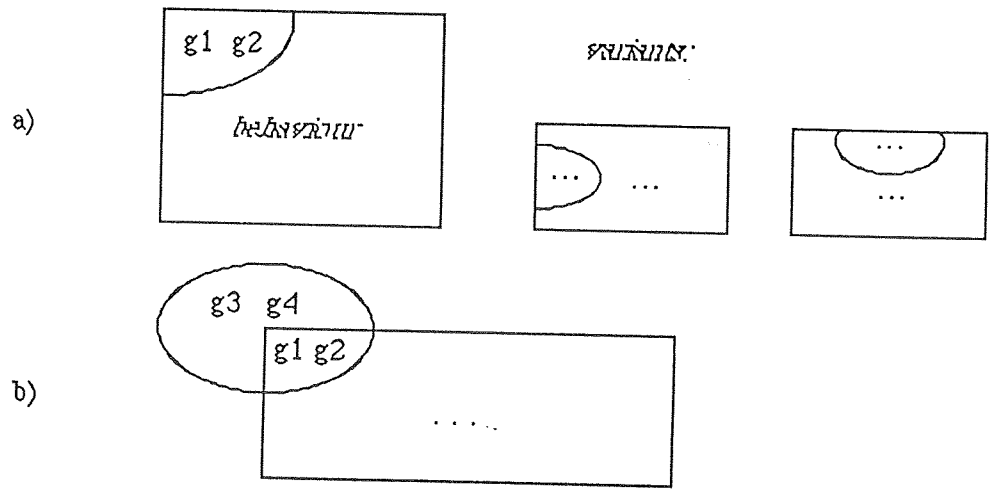


Figure 2.6

2.7 Guarding

The construct '[*E1 = E2*] -> *behaviour*', where [*E1 = E2*] is the condition (or 'guard') which enables the specified *behaviour*, is shown in Figure 2.7.

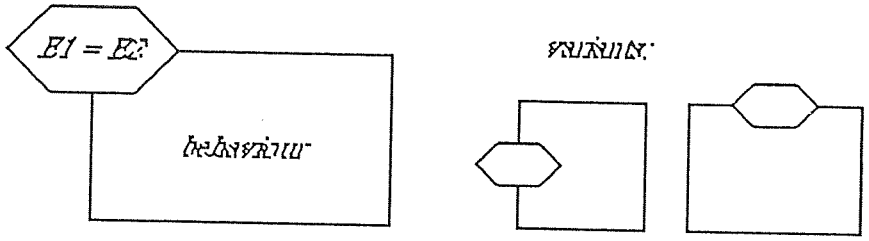


Figure 2.7

E1 and *E2* in figure denote the textual representations of two LOTOS value expressions. We deliberately avoid using an arrow symbol in the graphic representation of this construct, since arrows are preserved for denoting exclusively sequentiality of events and processes.

2.8 Parallel composition - interleaving

The parallel composition '*Behavior1 ||| Behaviour2*' of two behaviours which do not synchronize with each other is shown in Figure 2.8.

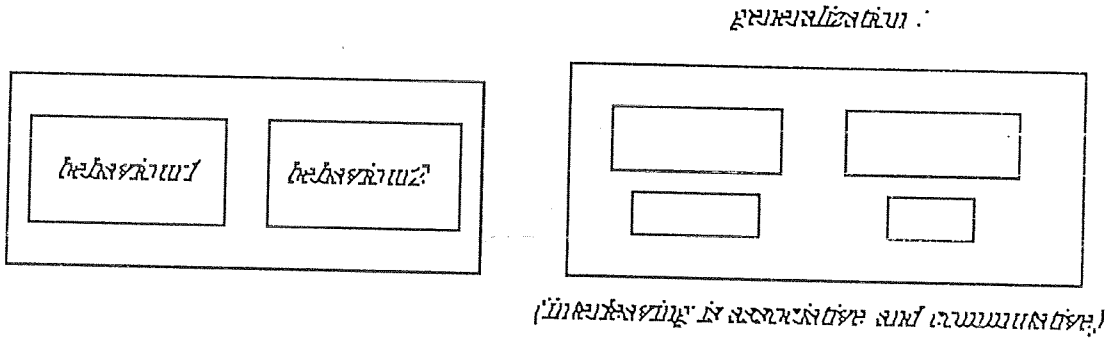


Figure 2.8

Suitable simplifications can be defined to eliminate some of the internal rectangles, without eliminating their contents.

2.9 Parallel composition - general case and full synchronization

The parallel composition '*Behavior1* |[g1, ..., gn]| *Behaviour2*' of two behaviours which synchronize at the synchronization gates [g1, ..., gn], and the parallel composition '*Behaviour1* || *Behaviour2*' of two behaviours which proceed in full synchronization are shown in Figure 2.8.

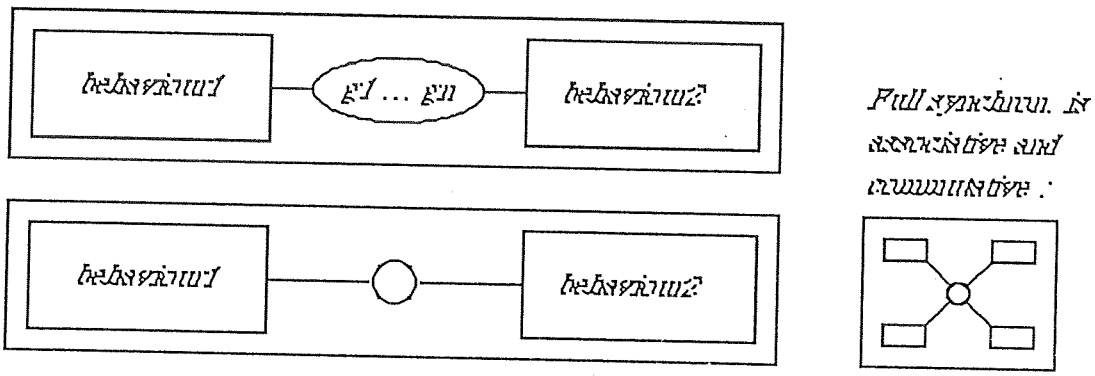


Figure 2.8.

Note that our formalization of the LOTOS graphic syntax cannot handle, currently, the syntactic abbreviation of Figure 2.8 for multiple application of the parallel operator.

2.10 Choice

The choice '*behaviour1* [] *behaviour2*' between two behaviours is shown in Figure 2.10.

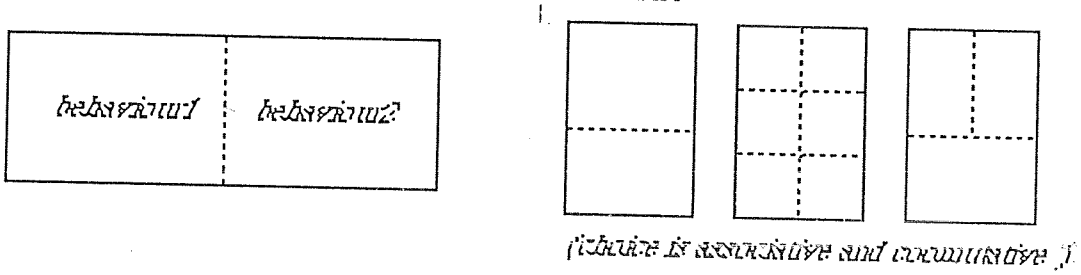


Figure 2.10

2.11 Local binding construct ('let')

The binding of variables x and y, or respective sorts sort1 and sort2, to the values of value expressions *E1* and *E2*, respectively, valid within a given *behaviour*, is achieved by the construct 'let x : sort1 = *E1*, y : sort2 = *E2* in *behaviour* '. Its graphic representation is given in Figure 2.11.

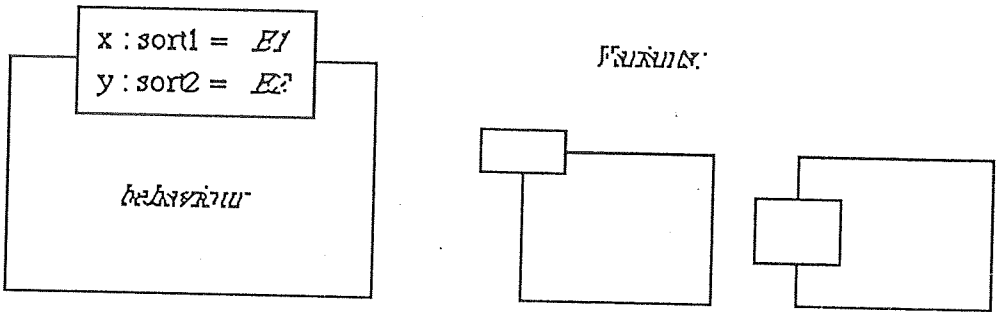


Figure 2.11

2.12 Multiple choice on values, multiple choice on gates

The construct 'choice x : sort1, y : sort 2 [] *behaviour*' expresses the choice of any instance of *behaviour* obtained by binding its variables x and y to values of sorts sort1 and sort2, respectively. Its graphic representation conveniently resembles that of the local binding construct, and is given in Figure 2.12. The analogous construct 'choice g in [g1, ..., gn], h in [h1, ..., hm] [] *behaviour*', where the alternatives are generated by different instantiations of gates g and h, is illustrated in Figure 2.12.b.

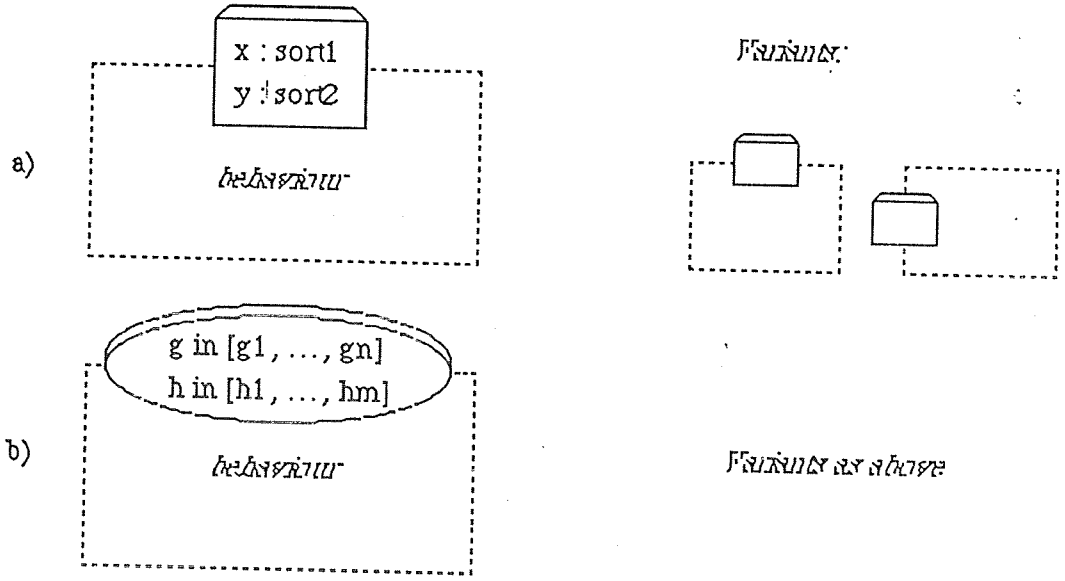


Figure 2.12

Note that dashed lines were used also for the (binary) choice construct.

2.13 Multiple parallelism ('par')

The construct

'par g in [g1, ..., gn], h in [h1, ..., hm] ||| *behaviour*'

expresses the parallel composition, without synchronization, of several instances of a specified *behaviour*, generated as for the multiple choice on gates of Section 2.12. In fact, any one of the

three cases of LOTOS parallel composition (interleaving, general case, full synchronization, see Section 2.9) can be applied here, and the corresponding graphic representations, given in Figure 2.13, are obvious adaptations of the graphical representations of those three cases.

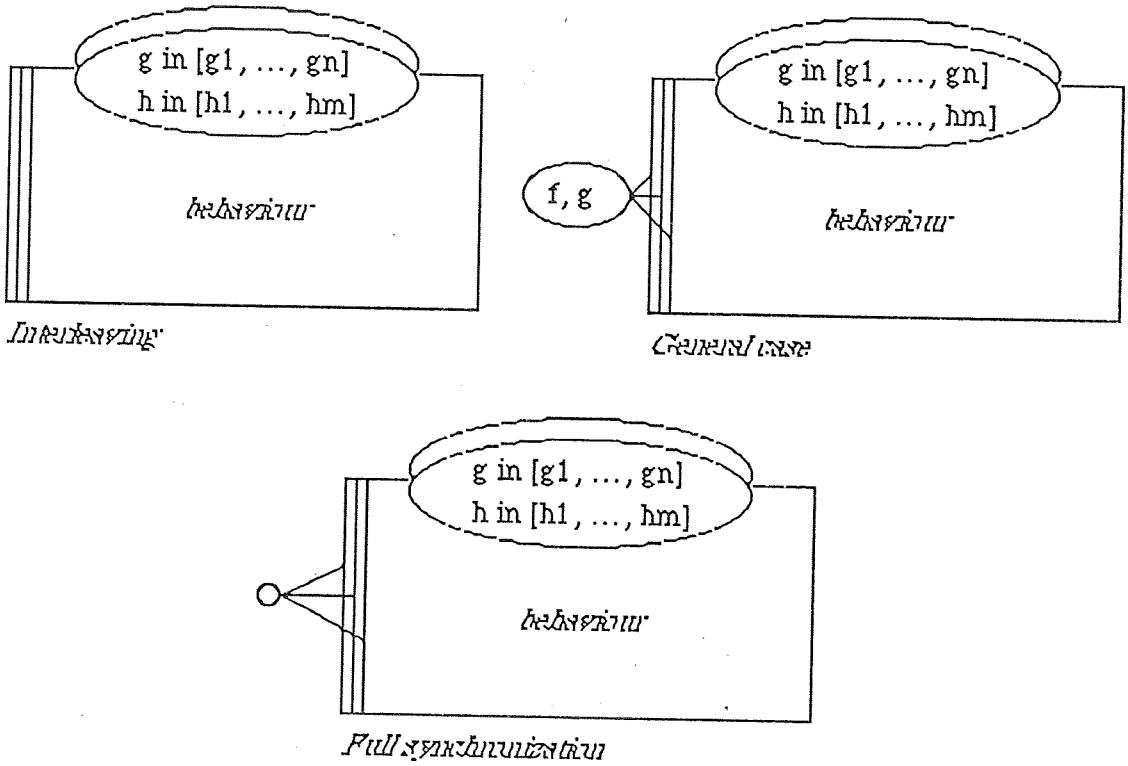


Figure 2.13

2.14 Enabling

In the sequential composition

behaviour1 >> accept x:sort1, y:sort2 in *behaviour2*

of two behaviours, at termination of the first behaviour two values of sorts sort1 and sort2 are bound to variables x and y, respectively, and used via these variables in the subsequent*behaviour2*. This construct is represented as shown in Figure 2.14.

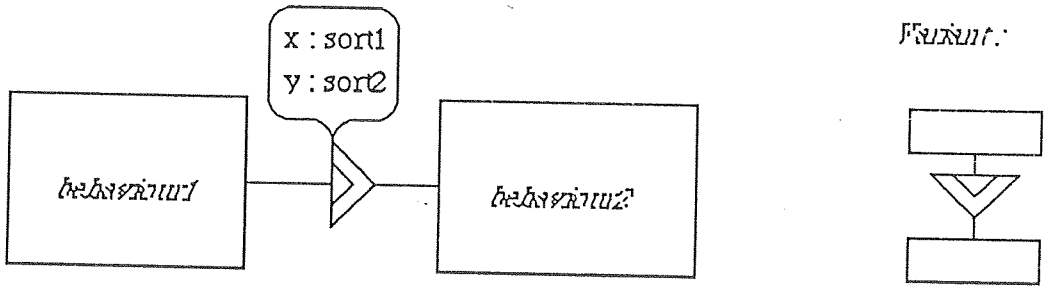


Figure 2.14

2.15 Disabling

In expression:

behaviour1 [> *behaviour2*

the first behaviour can be disabled by the second one. This is represented as in Figure 2.15.

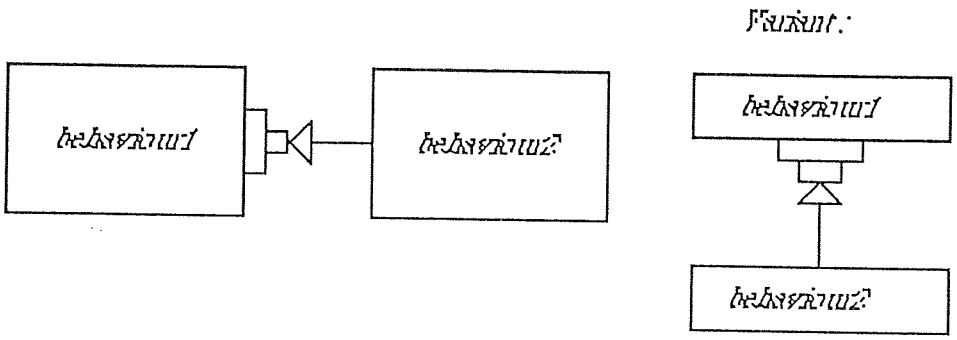


Figure 2.15

3. A COMPLETE SPECIFICATION

We give here the graphic version of the LOTOS specification of the Daemon_Game; its textual specification can be found in [BB88]. The original LOTOS specification of the Daemon_Game is due to Chan and Turner ([CT86]).

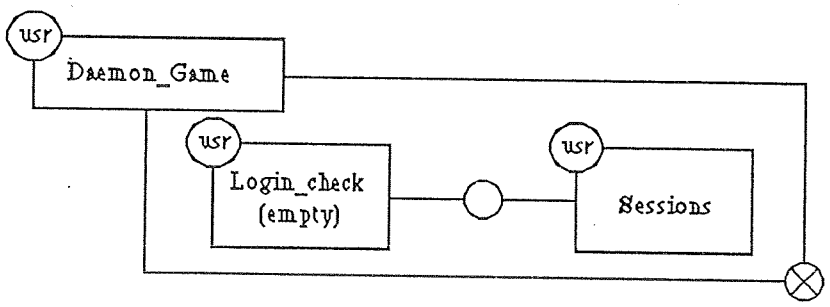
Boolean, Set
NaturalNumbers

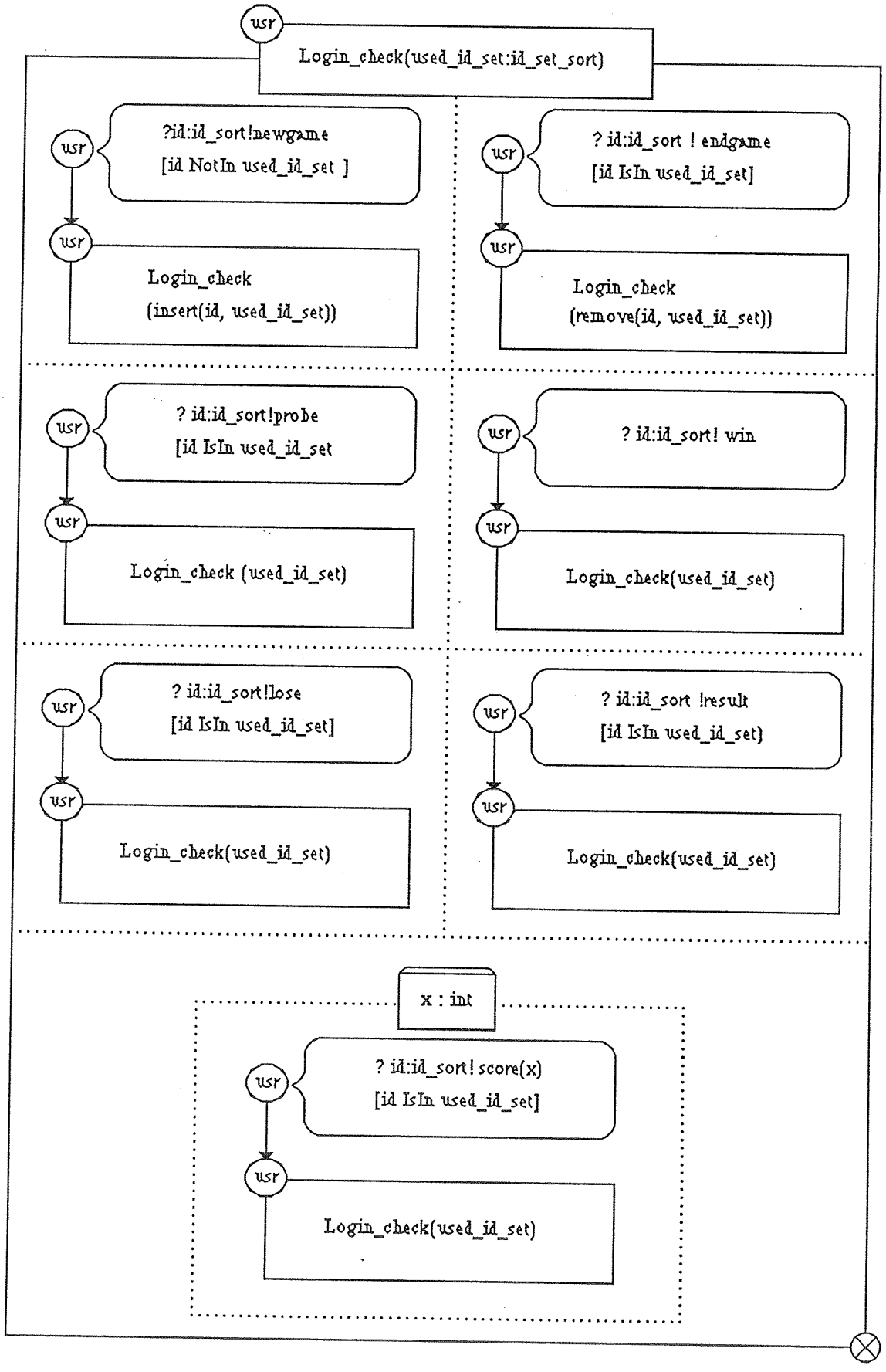
```
type Integer is
   sorts int
   opns 0          :  -> int
        inc, dec : int -> int
   eqns forall n :int
        ofsort int
          inc(dec(n))=n
          dec(inc(n)) = n
endtype
```

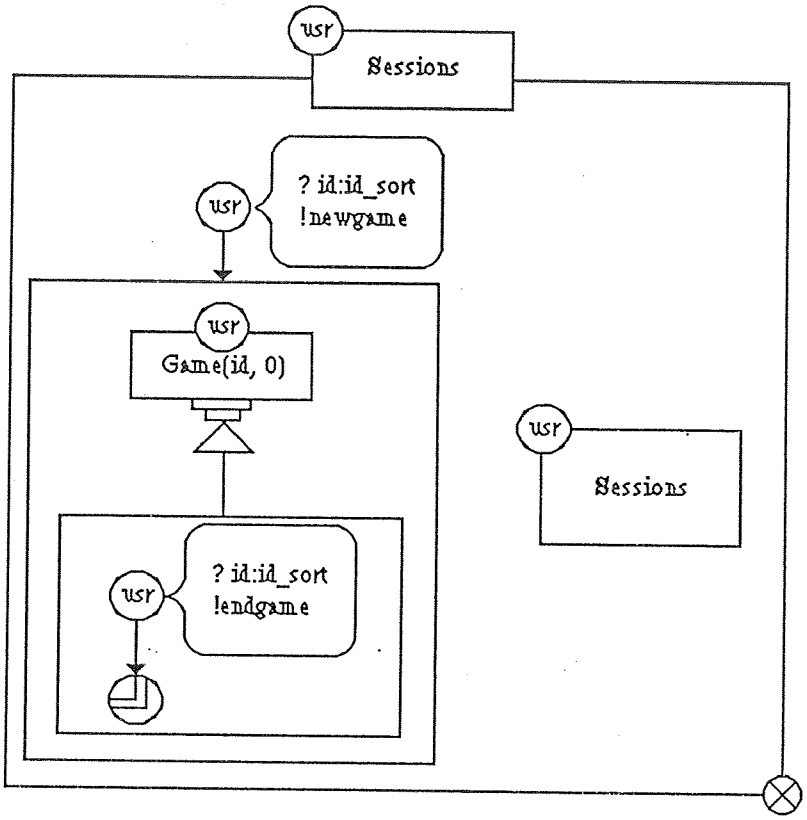
```
type Signal is
   sorts sig_sort
   opns newgame,probe,win,
        endgame,result,lose :    ->sig_sort
        score                    : int ->sig_sort
endtype
```

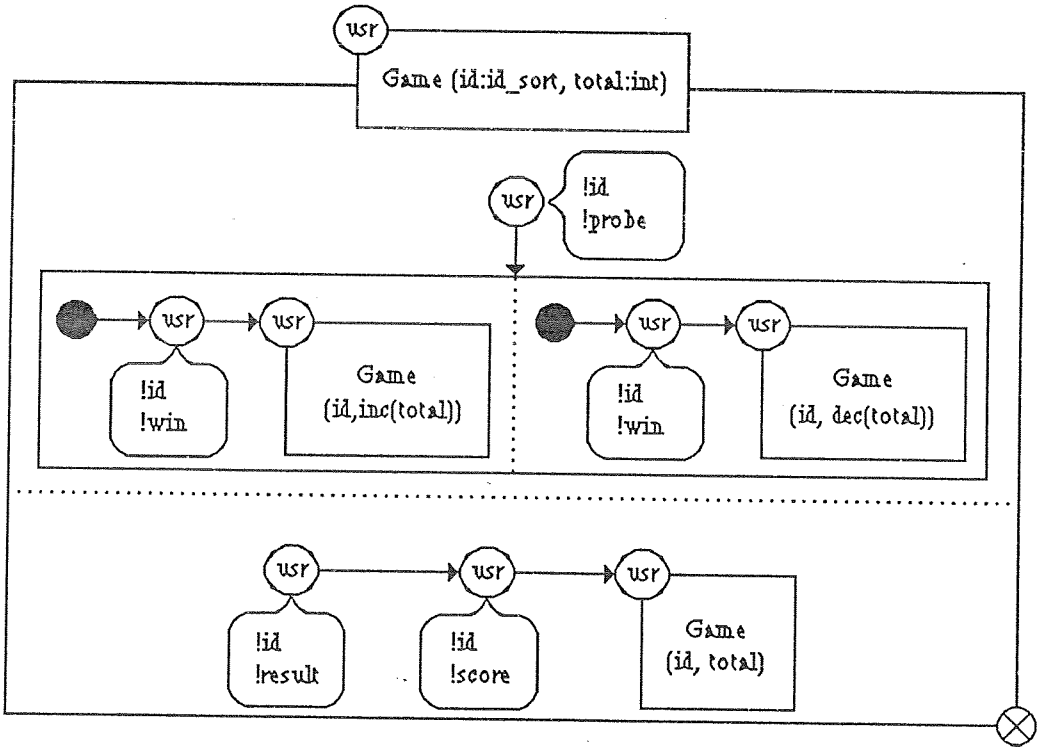
```
type Identifier is NaturalNumber
  renamedby
    sortnames
      id-sort for nat
endtype
```

```
type Identifier_set is Set
   actualizedby Identifier using
      sortnames
         id_sort      for elem
         id_set_sort for set
endtype
```




usr
Login_check(used_id_set:id_set_sort)
?id:id_sort!newgame
[id NotIn used_id_set]
Login_check
(insert(id, used_id_set))
? id:id_sort ! endgame
[id IsIn used_id_set]
Login_check
(remove(id, used_id_set))
? id:id_sort!probe
[id IsIn used_id_set
Login_check (used_id_set)
? id:id_sort! win
Login_check(used_id_set)
? id:id_sort!lose
[id IsIn used_id_set]
Login_check(used_id_set)
? id:id_sort !result
[id IsIn used_id_set)
Login_check(used_id_set)
x : int
? id:id_sort! score(x)
[id IsIn used_id_set]
Login_check(used_id_set)





4. Conclusions

We have introduced a graphic syntax for LOTOS. A formal scheme based on Prolog has been adopted in order to derive pictures from the abstract syntax of the language. One complete example of a graphic specification was given. Our current proposal does not deal with the nesting of process and type definitions (and associated scoping rules). Moreover, it does not provide graphical representations of type definitions and value expressions. We feel that the logic programming style adopted for the definition of graphical LOTOS offers a satisfactory level of formality and abstraction, and at the same time it provides a good basis for the implementation of graphic tools.

AKNOWLEDGMENTS

We wish to express our gratitude to Paul Tilanus, who started the activity on the graphic syntax for LOTOS, and to Elie Najm and Pippo Scollo, for discussions on this topic. The first author is grateful also to the University of Twente (and to Chris Vissers) for having partially supported his participation to this activity.

REFERENCES

[AS87] R. Ahooja, B. Sarikaya, "A Graphical Representation for LOTOS FDT", draft, 1987.

[BB88] T. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS", to appear on Computer Networks and ISDN Systems, North-Holland.

[B87] S. A. Black, "On a Graphical Interface for LOTOS", draft, Oct. 1987.

[BLMP87] T. Bolognesi, D. Latella, S. Mannucci, A. Pisano, "A graphical syntax for LOTOS specifications", draft, 1987.

[BT87] G. Bruno, S. Trigila, "A Proposal for a Graphical LOTOS", draft, 1987.

[CT86] W. F. Chan, K. Turner, "The Daemon Game in LOTOS", in: ESTELLE, LOTOS, SDL Draft Examples, Joint Meeting ISO/CCITT, Turin, December 1986.

[CCITT87] CCITT - Recommendation Z.100 - Specification and Description Language SDL, 1987.

[DOD80] U. S. Department of Defense: "Requirements for ADA Language Integrated Computer Environments", STONEMAN, 1980.

[GRA86] GRASPIN Project Team: "Architecture of the Final GRASPIN Workstation Prototype, GRASPIN Technical Paper GRA 80/2, October 1986. ISBN: 3-88457-920-7

[HT86] S. Horwitz, T. Teitelbaum, "Generating Environments Based on Relations and Attributes", ACM TOPLAS, Vol. 8, N. 4, pp. 577-608, 1986.

[ISO-Q48.4] ISO/TC97/SC21 N.2014, "Proposed new question on Graphical representations for LOTOS, Project 97.21.9 Q48.4, June 1987.

[ISO-8807] ISO/TC97/SC21 N.2152 (also ISO/DIS 8807), "Revised Text of 2nd ISO/DP 8807 - Information Processing Systems - Open Systems Interconnection - LOTOS - A

Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", August 1987.

[K85] D. S. Knotkin, "The Gandalf Project". The Journal of System and Software, Vol. 5, N. 2, May 1985.

[NQ87] E. Najm, J. Queiroz, "Towards Graphical Representations for LOTOS", draft, 1987.

[SS86] L. Sterling, E. Shapiro, *The Art of Prolog*, The MIT Press, 1986.

[T81] W. Taitelman, L. Massinter: "The Interlisp Programming Environment", Computer Magazine, April 1981.

[T87] P. Tilanus, "A proposed graphical syntax for LOTOS" - draft - 1987.

APPENDIX A - AN ABSTRACT SYNTAX FOR LOTOS

The importance of defining an *abstract syntax*, in dealing with a programming or specification language, is today widely recognized, and its use is a consolidated practice. Such a practice has steamed from the consideration that a program (or specification, but in the sequel we shall speak of "programs" in a rough sense) must not be considered simply as a string of characters or lines of text. Instead, it must be considered as a *structured* object, constituted by *constructs* which are meaningful from the point of view of the *semantics* of the language. The rules which define the structure of programs constitute the *abstract syntax* of the language. It is important to point out that such a syntactic description of the language doesn't deal with :

- the actual concrete representation of the various constructs when they are printed on paper or displayed on a screen;
- issues like precedence rules and/or associativity of operators of the language;
- ambiguities which my arise when "parsing" a particular representation of a program.

Issues like those mentioned above are left to other, more *concrete* representations. The interesting point is that any such representation, as well as any other interpretation, including semantic ones, can be defined as a mapping from the elements of the abstract syntax into the domains of interest. Actually, the abstract syntax management is the kernel activity of any integrated environment for software development ([DOD80,T81, GRA86, HT86, K85]).

The simplest and most intuitive way for describing the abstract syntax of a language consists of:

- grouping omogeneous constructs into classes (i.e. defining the set of *syntactic categories* of the language), and
- defining the structure of each structured construct in terms of component sub-constructs (i.e. stating what are the syntactic categories to which such sub-constructs belong).

The simple hierarchical relationship between a construct and its components may be easily represented by trees. So, for example, the fact that the LOTOS *parallel composition* construct is composed by

- the two behaviour expression components
- the list of the gates at which synchronization occurs (i.e. a list of identifiers)

may be suitably represented by the tree shown in Figure A.1.

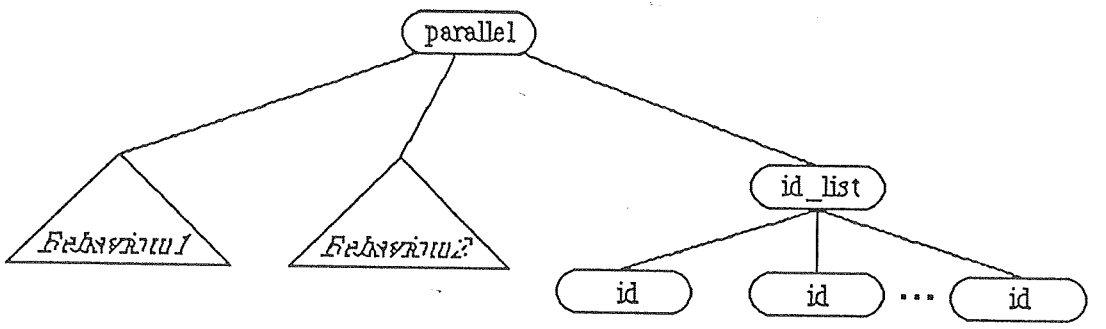


Figure A.1

Each node of the tree is labelled either with the name of the construct it represents (e.g. 'parallel', 'id_list', 'id'), in which case there will be a child for each subcomponent of the construct, or with the name of a meta-syntactic variable (e.g. *'Behaviour1'*, *'Behaviour2'*), ranging in some syntactc category (e.g.'behaviour expressions'). The tree in Figure A.1 does not represent a completely defined element of the language, because of the presence of variables; actually, only trees without variables represent completely defined instances of constructs. On the other hand the presence of variables in the abstract syntax trees is very useful when defining mappings of the abstract syntax into other domains. Indeed, in Appendix C we shall describe our graphic representation of LOTOS as a mapping of LOTOS abstract syntax into an abstract "world of pictures".

Let us now give an abstract syntax for LOTOS. It is defined as the language generated by a grammar, whose generic production rule looks like the following:

```
Synctactic_category   ->
        costruct_1
                (       variable_for_subconstruct_11        :       Synctactic_category,
                        ...
                        variable_for_subconstruct_1n        :       Synctactic_category
                )
        ...
|       costruct_m
                (       variable_for_subconstruct_m1        :       Synctactic_category,
                        ...
                        variable_for_subconstruct_mn        :       Synctactic_category
                )
```

In the above rule, Syntactic_category is a word starting with an upper case letter which represents a syntactic category; construct_i is a word composed by lower case letters only and construct_1, ..., construct_m represent the alternative constructs of the syntactic category defined by the rule. The metasymbol '|' has the usual "alternative" meaning. For each construct, the list of syntactic categories of its components is given; each element of the list is associated with a variable (in *italics*) which ranges in that category. In order to produce both completely defined and partially defined instances of the constructs, the *variable_for_subconstruct* and the Syntactic_category must be considered as disjoint alternatives (so, the metasymbol ':' is equivalent to '|'). Finally, some meta-syntactic abbreviation are used, for expressing optionality and for lists, whose meanings are obvious.

In the following we give a (partial) abstract syntax for LOTOS. Essentially, this grammar defines a language of *linearized* trees. For instance, 'parallel (Behaviour1, Behaviour2, id_list (id, id, ..., id)' is an element of this language, and represents the linearization of the tree in Figure A.1. It is worth noting that such a language is a subset of the language of Prolog terms [SS86].

```
Specification   ->
        specification
                (Spec_id:               Id,
                Gates:                  Optional_Nonempty_Id_List,
                Parameters:     Optional_Nonempty_Id_Decl_List,
                Functionality:  Optional_Id_List,
                Global_types:   Optional_Nonempty_Type_Def_List,
                Behaviour:              Beh_Expr,
                Local_Defs:     Optional_Nonempty_Local_Def_List)

Optional_<item>         ->
                void | <item>

Local_Def               ->
        type_def (...)
```

```
|       proc_def
            (Proc_Id:           Id,
            Gates:          Optional_Nonempty_Id_List,
            Parameters:     Optional_Nonempty_Id_Decl_List,
            Functionality:  Optional_Id_List,
            Behaviour:          Beh_Expr,
            Local_Defs:     Optional_Nonempty_Local_Def_List)

Beh_Expr            ->
        stop
|       exit    (Parameters:    Exit_Parameter_List)
|       proc_inst
            (Proc_Id:           Id,
            Gates:          Optional_Nonempty_Id_List,
            Parameters:     Nonempty_Value_Expression_List)

|       guarded
            (Guard1:            Guard,
            Behaviour:          Beh_Expr)

|       choice
            (Behaviour1:    Beh_Expr,
            Behaviour2:     Beh_Expr)
|       parallel
            (Behaviour1:    Beh_Expr,
            Behaviour2:     Beh_Expr,
            Synchr_Gates:       Id_List)
|       full_synchronization
            (Behaviour1:    Beh_Expr,
            Behaviour2:     Beh_Expr)
|       unobs_prefix
            (Behaviour:     Beh_Expr)
|       obs_prefix
            (Gate:          Id,
            Offers:             Offer_List,
            Guard:              Optional_Guard,
            Behaviour:          Beh_Expr)
|       hide
            (Gates:             Nonempty_Id_List,
            Behaviour:          Beh_Expr)
|       enable
            (Behaviour1:    Beh_Expr,
            Behaviour2:     Beh_Expr,
            Parameters:     Optional_Nonempty_Id_Decl_List)
|       disable
            (Behaviour1:    Beh_Expr,
            Behaviour2:     Beh_Expr)
|       local_binding
            (Assignements:      Nonempty_Assignement_List,
            Behaviour:          Beh_Expr)
|       multiple_choice_on_values
            (Generator:     Id_Decl_List,
            Behaviour:          Beh_Expr)
|       multiple_choice_on_gates
            (Generator:     Gate_Decl_List,
            Behaviour:          Beh_Expr)
```

| multiple_parallel
(*Generator:* Gate_Decl_List,
Behaviour: Beh_Expr,
Synchr_Gates: Id_List)
| multiple_full_synchronization
(*Generator:* Gate_Decl_List,
Behaviour: Beh_Expr)

APPENDIX B - FORMAL SPECIFICATION OF GLOTOS - basic definitions

In this appendix we shall characterize the pictures we have informally introduced in Section 2. First of all we need some preliminary definitions.

Picture instance: a *picture instance* is a triple (S, Back, Front), where

S	is a set of points on the cartesian plane, and
Back and Front	are two distinguished points of S.

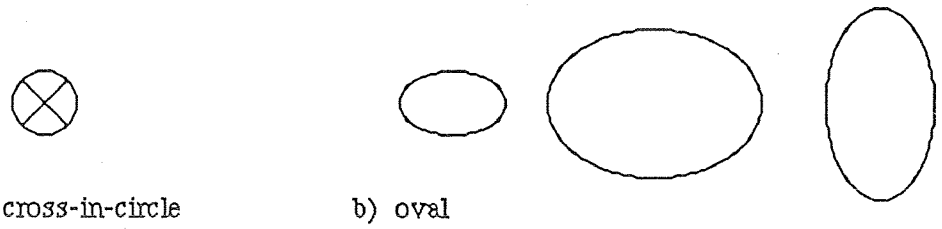
For convenience we will ambiguously use the word "picture (intance)" to refer both to the triple and to the set S (the intended meaning should be clear from the context).

Picture: a *picture* is a set of picture instances. We will define complex pictures in terms of a set of *basic pictures* and of a set of predicates which establish relations between pictures. Among basic pictures we distinguish between *constant-size* and *variable-size* pictures.

Constant-size basic picture: a *constant-size* basic picture is one whose instances can all be obtained by simply translating on the plane a unique picture instance.

Variable-size basic picture: any basic picture which is not a constant-size one.

The unique instance (modulo translation) of basic picture 'unit-cross-in-circle', of constant size, and three instances of basic picture 'oval', of variable size, are shown, respectively, in Figures B.1 (a) and (b).



a) unit-cross-in-circle b) oval

Figure B.1 - Constant and variable-size basic pictures.

Without getting involved into cumbersome geometric and topologic notions, we assume the existence of a number of unary, primitive predicates, named as the basic pictures, which may be satisfied by picture instances. Thus, picture P can now be defined as the set of picture instances which satisfy the associated predicate:

$$P = \{(S, Back, Front) \mid (S, Back, Front) \text{ is a picture instance, and } P(S, Back, Front)\}.$$

Predicates are useful since they allow one to characterize pictures at a convenient level of abstraction. Namely, by using predicates, we do not care about the *absolute* positions of picture instances in the cartesian space, or about their *actual* sizes. On the other hand, we shall impose precise constraints on pictures stated in terms of *relative* positions as well as *relative* sizes. Moreover, our "predicate" approach will allow us to express picture relations in a PROLOG-like style. The possibility offered by logic programming to define relations, which are a simple and way to express nondeterminism, proves quite useful also in order to include the notion of *variant* in the formal specification of GLOTOS. A further obvious advantage of using a PROLOG-like style is connected with rapid prototyping.

Figure B.2 lists our constant size picture predicates, and some picture instances which satisfy them. The distinguished points Back and Front are abbreviated 'b' and 'f': when they are not shown, their location is to be considered immaterial; 's' is the actual set of points of the picture instance.

Some instances of basic, variable size pictures are shown in Figure B.3. In the case of the partitioned_rectangle, n must be an integer and determines the number of component rectangles of the partitioned rectangle; h must be either the constant 'dashed' or the constant 'invisible'; in the first case the components of the partitioned rectangle will be separated by dashed lines as in the figure; in the second case, separation lines will be not visible. The predicate component (*pr, n, s, b, f*) is true if *pr* is a partitioned rectangle and *s* is the *n*-th component rectangle, whose back and front points are *b* and *f*.

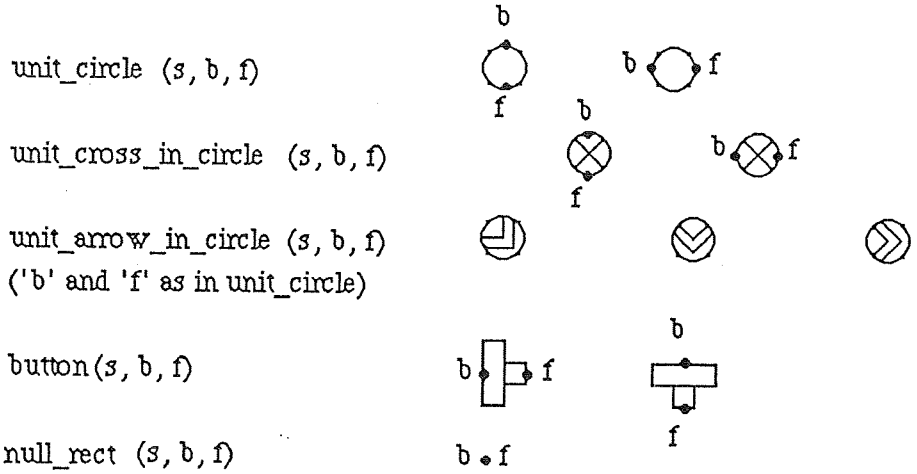


Figure B.2

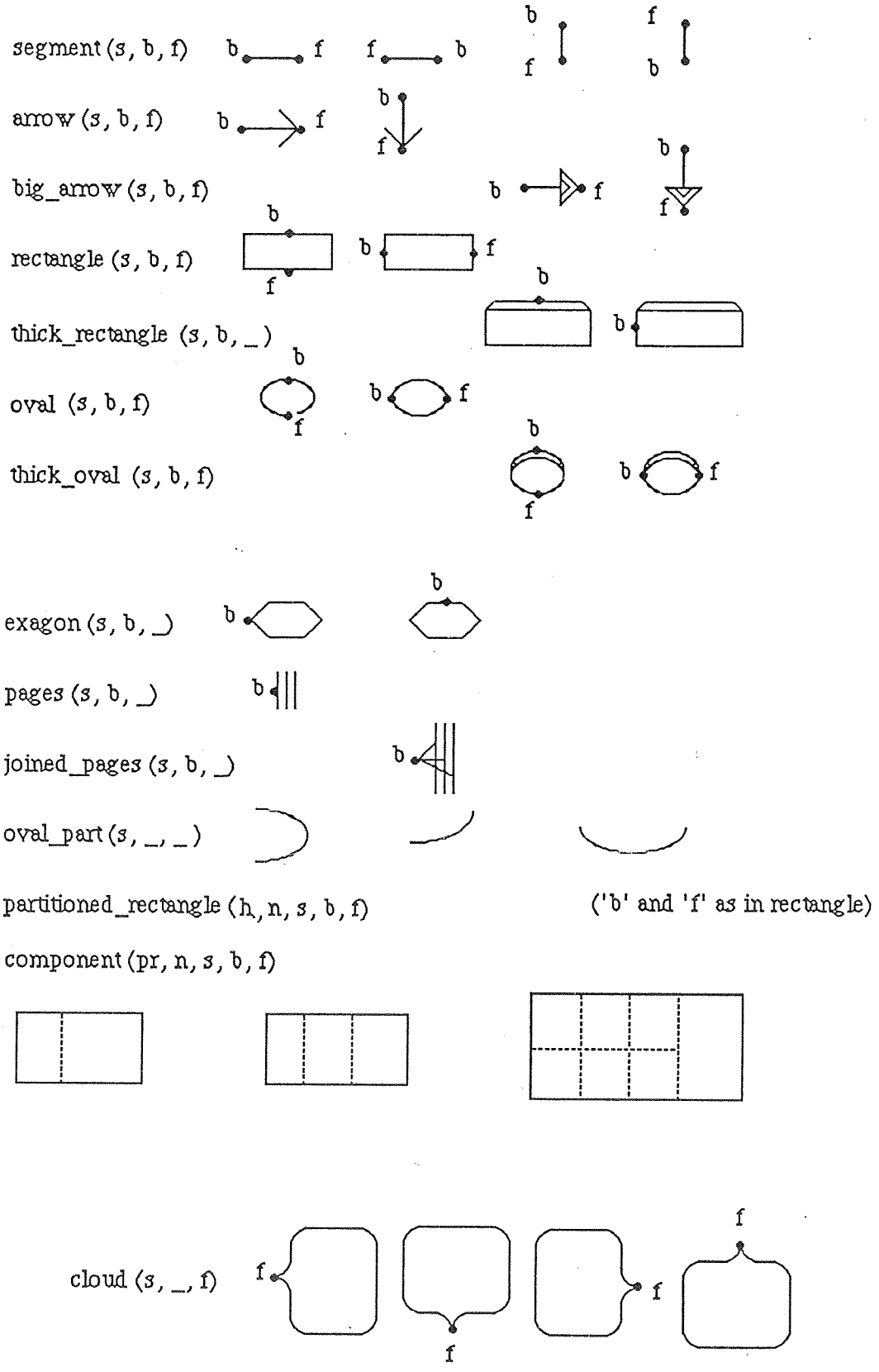

segment (s, b, f)
arrow (s, b, f)
big_arrow (s, b, f)
rectangle (s, b, f)
thick_rectangle (s, b, _)
oval (s, b, f)
thick_oval (s, b, f)
exagon (s, b, _)
pages (s, b, _)
joined_pages (s, b, _)
oval_part (s, _, _)
partitioned_rectangle (h, n, s, b, f)
('b' and 'f' as in rectangle)
component (pr, n, s, b, f)
cloud (s, _, f)

Figure B.3

Further predicates are introduced below. Together with the basic picture predicates listed above, these *primitive* predicates form the kernel our graphic system. In order to handle pictures conveniently, each one of them is associated with two invisible *reference rectangles* and eigthteen invisible *reference points* (besides the Front and Back). The two reference rectangles are identified by the two *reference rectangle identifiers* 'outrect' and 'inrect'.

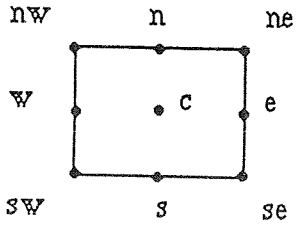
Figure B.4.a shows the nine invisible *reference-points* associated to an invisible reference-rectangle. They are identified by the *reference point identifiers'* n', 'ne', 'e', 'se', 's', 'sw', 'w', 'nw', 'c' (n for north, e for east..., c for center). Any subset of the reference points of either reference rectangle associated to a picture can be identified by the primitive predicate 'ref_point':

ref_point (Picture, Ref_rect_id, Ref_pids, Ref_points) if
Ref_points is a list of points of the reference rectangle Ref_rect_id ('inrect' or 'outrect') associated with Picture; these points are identified by the reference point identifiers in list Ref-pids.

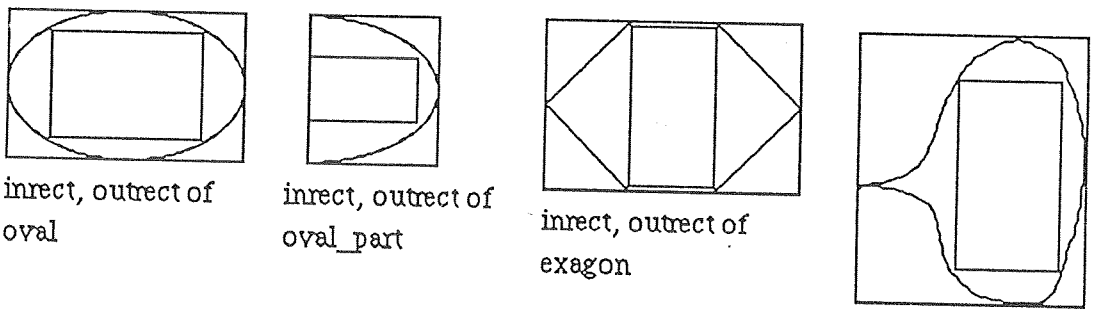
Reference rectangles can be obtained by the primitive predicate 'ref_rect':

ref_rect (Picture, Ref_rect_id, Ref_rect) if
Rectangle Ref_rect is the minimum rectangle which delimits Picture or a maximal rectangle included in the area delimited by Picture, when Ref_rect_id is, respectively, 'outrect' or 'inrect'.

The invisible reference rectangles of some basic pictures are shown in Figure B.4.b, which covers the most peculiar cases; in all other cases the two rectangles are the obvious ones (most times they coincide).



(a)



(b)

Figure B.4 - Reference rectangles and points.

A special class of pictures is represented by texts, which can be obtained by the predicate 'text'

text ([Term1, ..., Termn], Pict) if
Pict is a picture representing the concatenation of the texts denoted by the terms [Term1, ..., Termn]. The null rectangle represents the empty list.

Of course the 'outrect' rectangle is defined also for text-pictures. Four more primitive predicates are useful for building complex pictures: they are listed below.

union ([Pict1, ..., Pictn], Pict) if
(the set of points of) picture Pict is the union of (the sets of points of) pictures Pict1, ..., Pictn.

disjoint ([Pict1, ..., Pictn]) if
the intersection of the areas delimited by the 'outrect' rectangles of pictures Pict1, ..., Pictn is empty, or is a null area.

difference (Pict1, Pict2, Pic1') if
Pict1' is the set of points of picture Pict1 not included in the area delimited by the 'outrect' of picture Pict2.

rect_contains (Rect1, Rect2) if
the area delimited by rectangle Rect2 is included in the area delimited by rectangle Rect1.

Finally, we will use the two derived predicates:

```
contains (Pict1, Pict2) if
        ref_rect (Pict1, inrect, Rect1),
        ref_rect (Pict2, outrect, Rect2),
        rect_contains (Rect1, Rect2)
        refpoint (Pict1, [c], P1), refpoint (Pict2, [c], P2), common_points (P1, P2).
common_points (Point_list1, Point_list2) if
        on (X, Point_list1), on (X, Point_list2).
```

where on (X, L) if *X is an element of the list L.*

APPENDIX C - FORMAL SPECIFICATION OF GLOTOS - Unparsing mapping

The concrete graphic syntax for LOTOS proposed in this paper is defined formally by providing a mapping from abstract syntax terms (see Appendix A) into pictures (built as discussed in Appendix B). More precisely, the unparsing is given as a Prolog program which defines predicate 'picture'. Informally:

picture (Term, Pict, Back, Front) if
the set of points Pict, with distinguished points Back and Front, is the picture associated with the abstract syntax term Term.

Pictures are defined by induction on the structure of the terms, thus the clauses of predicate Picture will be, in general, recursive. As usual, variables names start with upper-case. The unparsing given here does not cover the whole abstract syntax of Appendix A, but is sufficient to illustrate the technique.

```
picture (proc_def (Proc_Id, Gates, Parameters, Functionality, Behaviour, Local_Defs), Pict, _, _ )
        if
                oval (O, _,_), rectangle (R1, _, _), rectangle (R2, _, _),
                text ([Gates], O_text), contains (O, O_text),
                text ([Proc_Id, Parameters], R1_text), contains (R1, R1_text)
                picture (Behaviour, B, _,_ ), contains (R2, B),
                func_symbol (Functionality, F_sym),
                func_cloud (Functionality, F_cloud, F_front),
                func_sorts (Functionality, F_sorts), contains (F_cloud, F_sorts),
                ref_points (O, outrect, [c], O_center),
                ref_points (R1, outrect, [w,n,nw], R1_points),
                ref_points (R1, outrect, [c], R1_center),
                ref_points (R2, outrect, [w,n,nw], R2_points),
                ref_points (R2, [sw], R2_sw),
                ref_points (F_sym, outrect, [c], F_center)
                ref_points (F_sym, outrect, [e,s], F_points),
                common_points (O_center, R1_points),
                common_points (R1_center, R2_points),
                common_points ( R2_sw, F_center),
                common_points ( F_points, F_front),
                disjoint ([ O, R1_text, R1, B, F_cloud]),
                difference (R1, O, R1_m_O),
                difference (R2, R1, R2_m_R1),
                difference (R2_m_R1, F_symbol, R2_m_R1_F),
                union ([O_text, O, R1_m_O, R1_text, R2_m_R1_F, B, F_sym, F_cloud,
F_sorts,Pict)

picture (stop, Pict, Back, Front)
        if
                unit_cross_in_circle (Pict, Back, Front).

picture (exit ([]), Pict, Back, _ )
        if
                unit_arrow_in_circle (Pict, Back, _ ).

picture (exit (Parameters), Pict, Back, _ )
        if
                Parameters ≠ [],
                unit_arrow_in_circle (Ci, Back, _), cloud (Cl, _, Cl_front), text([Parametrs],
Cl_text),
```

```
                contains (Cl, Cl_text),
                ref_points (Ci, outrect, [n, s, e, w], Ci_points),
                common_points ([Cl_front], Ci_points),
                disjoint (Ci, Cl)
                union ([Ci, Cl, Cl_text],  Pict).

picture (unobs_prefix (Behaviour), Pict, Back, Front)
        if
                black_unit_circle (C, Back, X), arrow (A, X, Y), picture (Behaviour, B, Y, Front),
                disjoint (A, B, C),
                union ([A, B, C], Pict).

picture (obs_prefix (Gate, Offers, Guard, Behaviour), Pict, Back, Front)
        if
                oval (O, Back, X), arrow (A, X, Y), picture (Behaviour, B, Y, Front),
                cloud (Cl, Cl_front)
                ref_points (O, outrect, [n, s, e, w], O_points),
                common_points ([Cl_front], O_points)
                text ([Gate], O_text), text ([Offers, Guard], Cl_text),
                contains (O, O_text), contains (Cl, Cl_text),
                disjoint (O, A, Cl),
                union ([O, O_text, Cl, Cl_text, A], Pict).

picture (proc_inst(Proc_Id, Gates, Parameters), Pict, Back, Front)
        if
                oval (O, Back, _), rectangle (R, _, Front),
                text ([Gates], O_text), contains (O, O_text),
                text ([Proc_Id, Parameters], R_text), contains (R, R_text),
                ref_points (R, outrect, [w, nw, n], R_points], ref_points (O, outrect, [c], O-points),
                common_points (R_points, O_points),
                disjoint (O, R_text),
                difference (R, O, R'),
                union ([O, O_text, R', R_text], Pict).

func_symbol ( void, FS)
        if
                unit_cross_in_circle (FS).

func_symbol ( Fuctionality, FS)
        if
                Functionality ≠ void, unit_arrow_in_circle (FS, _, _).

func_sorts (void, F_sorts)
        if
                null_rect (F_sorts, _, _).

func_sorts (Functionality, F_sorts)
        if
                Functionality ≠ void, text (Functionality, F_sorts).

func_cloud (void, F_cloud,F_front)
        if
                null_rectangle (F_cloud, _, F_front).

func_cloud (Functionality, F_cloud,F_front)
        if
                Functionality ≠ void, cloud(F_cloud, _, F_front).
```