SICS/R-89/8917

A Graphical G-LOTOS Editor Defined by the Meta-Tool LOGGIE

by

Björn Backlund   Tommaso Bolognesi
Olof Hagsand   Diego Latella
and Björn Pehrson

SICS research report
R89017
ISSN 0283-3638

A Graphical G-LOTOS Editor Defined by the Meta-Tool LOGGIE

Björn Backlund[1] - Tommaso Bolognesi[2] - Olof Hagsand[2]
Diego Latella[1] - Björn Pehrson[2]

Abstract

We present a prototype implementation of a graphical editor for a subset of G-LOTOS. The graphical G-LOTOS syntax is being standardized within ISO for the LOTOS specification language. First, we provide a formal definition of the G-LOTOS subset. Then, the specification of the editor is derived from this formal definition. Finally, the editor is generated automatically using the LOGGIE meta-tool.

1. Introduction

The advantages of adopting Formal Description Techniques (FDT's) for the specification of computing systems have been widely recognized within the community of system designers and developers, and also within standardization organizations, such as ISO and CCITT. In 1989, LOTOS, an FDT for communication services and protocols, has been introduced as an International Standard, [ISO89], [BB87]. FDT's allow you 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 a considerable effort is required to learn and understand them. In order to achieve a wide-spread acceptance of FDT's, they must be easy to learn and use, and must be supported by automated tools. One way to achieve this is to embed them in languages with intutive graphic syntax.

G-LOTOS is a graphical syntax for LOTOS currently being developed within ISO. The purpose is is to make LOTOS easy to learn and use. The graphical representation offered by G-LOTOS is meant to provide a "first glance" recognizability of the main features of the formally described system. The main features include temporal sequencing of events, alternative or parallel system behaviours, and synchronization. The work on G-LOTOS started in 1987 as a joint ISO/CCITT effort, and is still ongoing. The successive evolutions of various proposals for the concrete G-LOTOS symbols, and for the formalization of the graphical syntax, can be found in the following documents; [ISO87], [ISO88a], [ISO88b], [ISO88c], [BLP88], [BL89a], [BL89b]. The G-LOTOS standard is expected to be issued by ISO in 1991.

In the present paper, we describe our experience in implementing a prototype for a *language based graphical editor* for a subset of the current G-LOTOS proposal. The prototype was developed to experiment with the current proposal in order to assess it, and to get a more concrete understanding of the usefulness of possible forthcoming G-LOTOS tools.

1) SICS - Swedish Institute of Computer Science - BOX 1263 - S 164 28 KISTA - SWEDEN

2) C.N.R. Ist. CNUCE - Via S. Maria 36 I56126 Pisa - ITALY

The considered subset of G-LOTOS includes the following basic LOTOS constructs (no data types or data values are considered): process definition, process instantiation, action prefix (observable and unobservable), choice, enabling, parallell composition, stop, exit. Besides the user-oriented issues just mentioned, our interest in carrying out such an experiment was attracted by, and has concentrated upon:

i) the methodological aspects involved in the formalization of the graphical syntax, and
ii) the technological aspects, that is, the software engineering techniques and tools involved in the prototyping of the graphical editor.

In fact, the paper is primarily concerned with these two aspects.

The first step of our work was to give an *abstract* formal definition of the graphical syntax. Such a specification is "abstract" in the sense that it defines G-LOTOS as a set of abstract, mathematically well defined objects, namely sets of points in a Cartesian Space, abstracting from details like, for instance, actual picture sizes. The definition of G-LOTOS has been given by means of an abstract syntax for LOTOS and an algebraic relation which associates abstract syntax terms with sets of points. Such a specification has then been *actualized* into the specification of the editor itself, by introducing the definition of the interactions the user can have with the pictures displayed on the screen; these include cutting and pasting fragments of the graphical document, display of the fundametal LOTOS construct of 'process definition' associated with a user-selected occurrence of a 'process instantiation', and so on. From such a specification, the editor itself has been *automatically generated* using the meta-environment LOGGIE [BHP89] .

The use of meta-techniques for the automatic generation of language-based environments has been widely recognized as a valuable approach to software engineering [HN87]. LOGGIE is a meta-tool supporting generation of *interactive* graphical editors for languages with graphical syntax. An editor is generated by feeding the meta-tool with a *specification* of a language. Such a specification defines the abstract syntax extended with links (*garlands*) and *attributes* with attribute equations. One role of attributes is in the graphical presentation; they define how abstract syntax terms are associated with actual pictures.

LOGGIE extends the state of the art defined by well-known language-based meta-systems such as Cornell Synthesizer Generator [RT89] and Centaur [BCD88], by introducing an extended meta-notation and allowing the abstract syntax to go beyond the linear string structure of text-based languages, and the concrete syntax to include graphical symbols. LOGGIE generates interactive editors in a user-friendly environment supporting the editing of graphical structures in a syntax directed way. LOGGIE is layered in order to separate program synthezis and graphical structured presentation. The Derivation Layer handles the design and interpretation of specifications and provides the meta-user with an interactive graphic interface. The Structure Presentation Layer allows to associate different 'abstract' pictures (abstract graphical data types) to nodes of abstract syntax structures and to map them into concrete pictures on the screen. LOGGIE and the editors generated runs under the Xerox AI Environment [XAIE].

LOGGIE uses *attribute grammars* (AGs) for describing language semantics and presentation of abstract syntax structures. Attribute grammars is a formalism introduced in [Knuth68] to define the semantics of context free languages. Historically, AGs have been used as a tool to describe syntax-directed computations, such as compilers and translators. Recently, AGs have proved to be useful for interactive graphical applications such as the Higgens [H86] and GIGAS [FZ89] systems. Environments based on AGs can use incremental attribute evaluation techniques [Reps83], which is of great use in interactive language-based editors, such as the ones generated by LOGGIE, since only affected attributes are reevaluated after a change.

In Section 2, an informal, user-oriented description of the editor is given; In Section 3, we give an overview of the definition of the G-LOTOS syntax. In Section 4, the LOGGIE specification of the editor is presented. Finally, some concluding remarks are given in Section 5. Some familiarity with LOTOS is assumed, since the G-LOTOS pictorial representations will often be introduced by referring back to their textual LOTOS equivalent.

2. The G-LOTOS Editor generated by LOGGIE

In this section the G-LOTOS editor is introduced by showing an example of a typical editing session: the construction of a G-LOTOS process definition. The process will be defined by the user in a top-down fashion, starting with empty templates and continuing with a stepwise refinement of the components. The pictures provided in this section are snapshots from the screen.

The example we have used is the graphical equivalent of the LOTOS process definition given in Figure 2.1.

```
process T[a,b]:noexit :=
        P[a]|[a]|Q[a,b]
   where
        process P[g]: noexit :=
                g;P[g]
        endproc
        process Q[f,h]: noexit :=
                P[f] [] h;stop
        endproc
endproc
```

Figure 2.1: A process definition in LOTOS

We shall start by defining process T. We will then define the hierarchy, i.e. processes P and Q as local process definitions of T. Finally P and Q will be defined.

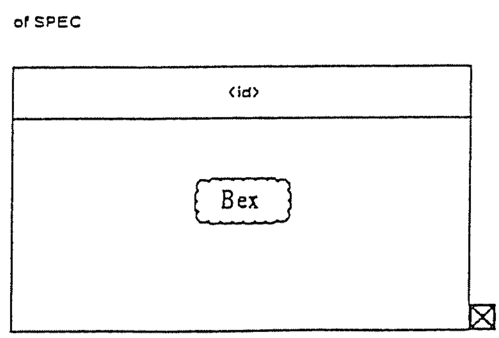


Figure 2.2 An unexpanded process definition frame

We start with the empty template of a process definition, illustrated in Figure 2.2. The big enclosing box represents the process definition itself. The upper part of the box is meant to eventually enclose the *name* of the process; in the picture it contains the unexpanded terminal <*id*>. The small crossed box in the lower right corner is the G-LOTOS symbol for the *noexit* process functionality, and is provided by the editor as the default functionality. The text occuring above the box, 'of SPEC', indicates the *path* of the process, i.e. the context in which the process is defined. In this example the process definition is imagined to appear at the outer level of the hierarchy of nested process definitions (of a textual LOTOS specification), thus its context is simply the specification itself. Finally, the *cloud symbol* is a meta-symbol which stands for an unexpanded *behaviour expression.*

The user can interact with the picture just by pointing at it with the mouse and invoking suitable operations. This is done with the help of pop-up menus, text editors etc. For example, the process identifier can be modified either by directly pointing at it and editing it, or by choosing '*edit name*' from the commands associated with the process definition frame.

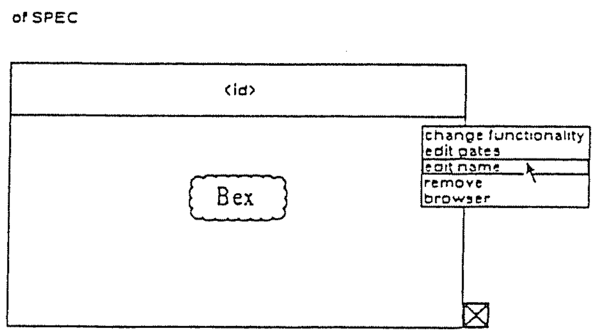


Figure 2.3 Changing the name of the process definition

Figure 2.3 shows the menu of the process definition and how the '*edit name*' operation is chosen. This brings up a text editor where the identifier can be provided by the user. Alternatively, one may point directly to the <id> symbol and type in the actual process identifier. The result of such an operation is shown in Figure 2.4

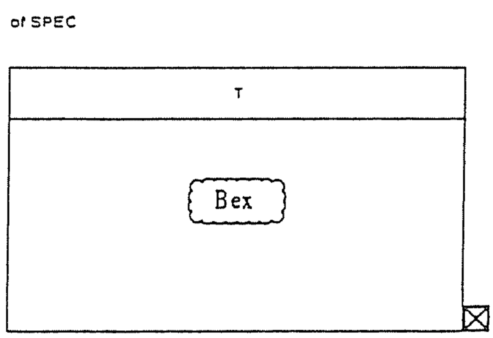


Figure 2.4 A process definition with an identifier

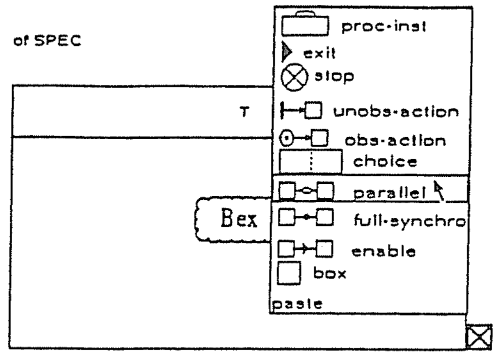


Figure 2.5 Interaction with the cloud

The behaviour expression is expanded via a menu which is displayed by clicking on the cloud symbol (Figure 2.5). The menu lists the alternative ways of expanding the behaviour, i.e. the behaviour operators defined in our subset of G-LOTOS. In the example of Figure 2.6 the user has chosen the *Parallel* operator: the system displays the two components of the parallel composition as unexpanded behaviours (again, the cloud meta-symbol), enclosed in a box.

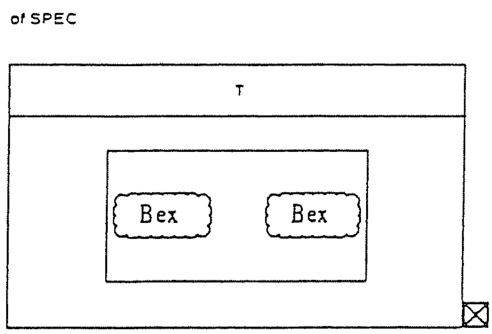


Figure 2.6 Parallell composition with no synchronization gates

The user may now operate on the parallel composition by pointing at the newly introduced box, thus visualizing another menu (Figure 2.7). Such a menu consists of a standard part, common to all behaviour expressions, which supports routine editing operations like cutting, copying, (this part is called *standard menu* and is further described in Section 4), and of a list of operator-specific commands.

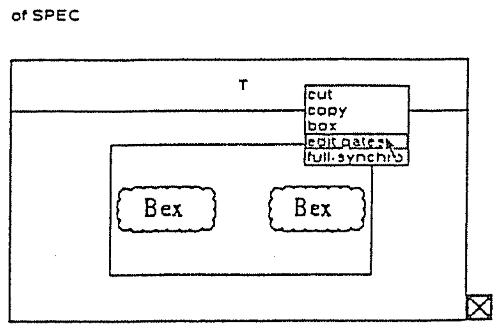


Figure 2.7 Selecting *edit gate* from the standard menu

The next step is to define the synchronization gate of the two (unexpanded) behaviours, namely a. This is done by choosing '*edit gate*' from the displayed menu and typing in the gate identifier. The resulting picture is shown in Figure 2.8.

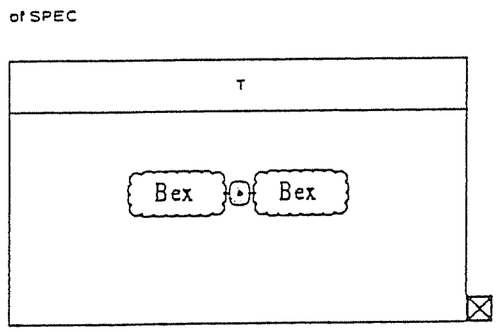
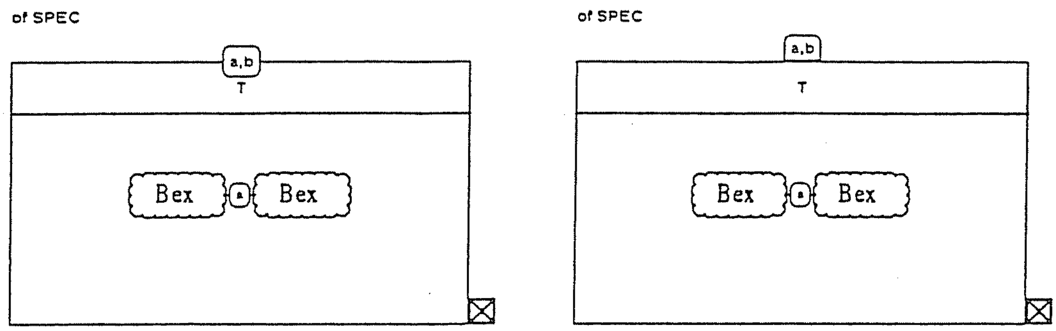


Figure 2.8 Synchronization on a, a undefined

Note that gate a is undefined in the context of the parallel behaviour, and is therefore shaded. This is an example of the use of marking to indicate a violated static semantics rule. Such rule can be satisfied by adding gate a to the formal gate list of the process definition. Again this is done by interacting with the process definition frame and by choosing '*edit gate*' from the menu of Figure 2.3.



(a) (b)

Figure 2.9 Formal gates assigned to process T

The result of adding gates *a* and *b* to the definition of process T is shown Figure 2.9 (a). The formal gates appear in a rounded box in the upper part of the process definition frame. In accordance with the current G-LOTOS proposal, the editor also supports alternative *aspects* of presentation. For example, the formal gate list has an alternative representation which is shown in Figure 2.9 (b).

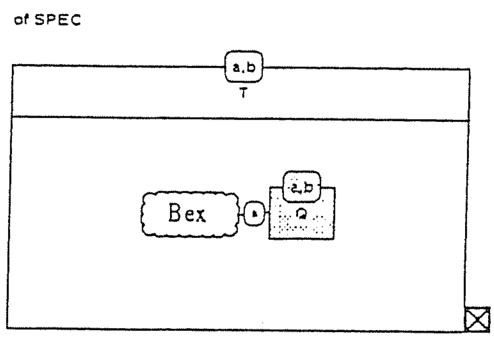


Figure 2.10 A process instantiation box

In Figure 2.10 the right-hand side behaviour has been expanded into the process instantiation Q[a,b]. The list of actual gates is placed in the rounded box in the upper part of the process instantiation box, just as in the case of the process definition. At the present,.there is no definition for Q, therefore the instantiation of Q is shaded. This is similar to the case of issuing a function call in a programming language where the function is undefined. In order to define Q, the user has to enter the *process browser* by selecting *browser* from the menu of Figure 2.3 .

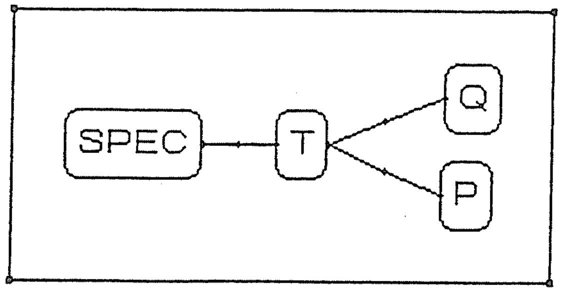


Figure 2.11 The process definition browser

Figure 2.11 shows a snapshot of the process browser illustrating an overview where the root node corresponds to the specification, SPEC, and the other nodes to process definitions T, P and Q. For example, Q is a local process definition of T and is therefore a child of T in the browser. Each node in the browser corresponds to a definition; actually the browser node and the definition are two aspects of the same object. When the user adds a node in the browser, a template of a process definition (see Figure 2.2) is created. The definition of the node may then be revealed by *opening* the browser node. Processes can be created, removed, copied and pasted freely within the browser.

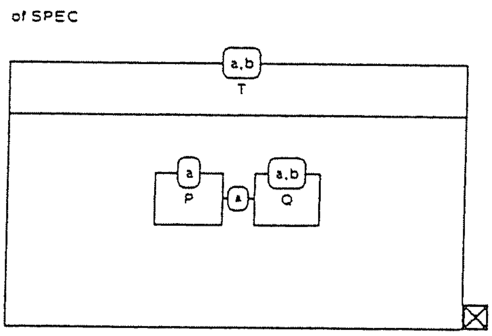


Figure 2.12 The complete process definition T

The complete definition of process T is given in Figure 2.12. Process T is a parallel composition of the instantiation of the two processes P and Q synchronizing at gate a(on a-actions).

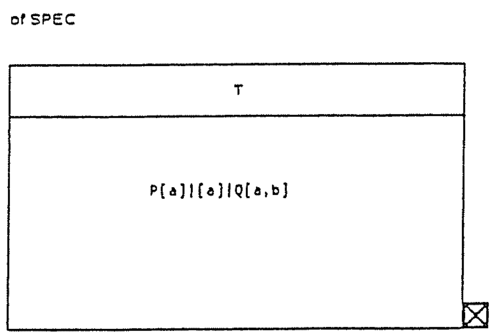


Figure 2.13 Displaying T behaviour as text

We can now choose to display the behaviour of T in textual LOTOS. This is done by selecting the *text* aspect of the behaviour as shown in Figure 2.13.

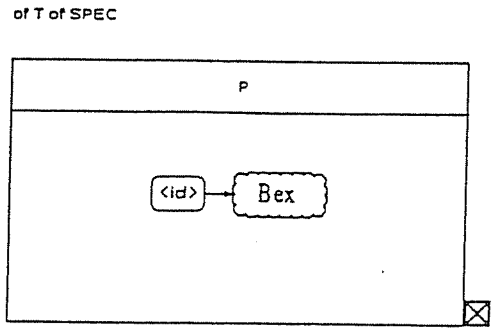


Figure 2.14 An unexpanded action prefix

Figure 2.14 shows a first shot of the definition of P, where the user has chosen to define the topmost behaviour of P as an *action prefix*: the action that will result from the expansion of the nonterminal <id> is to be followed in time (arrow symbol) by the behaviour that will result from the expansion of the cloud nonterminal. Again, the action is defined either by choosing from the standard menu associated with the *action prefix* construct, which is displayed by pointing to the connection arrow, or by pointing directly at the <id> symbol. Figure 2.15 shows the complete definitions of P and Q. The behaviour of P consists of action *g* followed by the instantiation of process P itself; hence, P is the unbounded sequence of *g*-actions. Process Q is defined as the nondeterministic choice between P and the action *h* followed by the *stop* process.

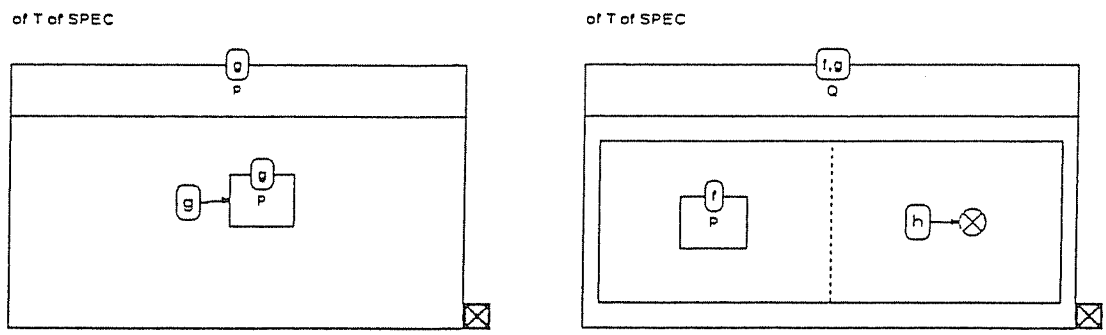


Figure 2.15 Processes P and Q

We have illustrated the phases of an editing session for creating a simple G-LOTOS specification. In practice, the G-LOTOS diagrams are more complex than those introduced above and may involve several layers of nested boxes. To be able to handle such cases conveniently, the editor supports the *folding* of any behaviour expression into an icon. An icon has the appearance of the construct it stands for, but does not include all the internal details of it, and is therefore much smaller. A folded icon can be *unfolded* into the full representation, either in place of the icon or inside a new window.

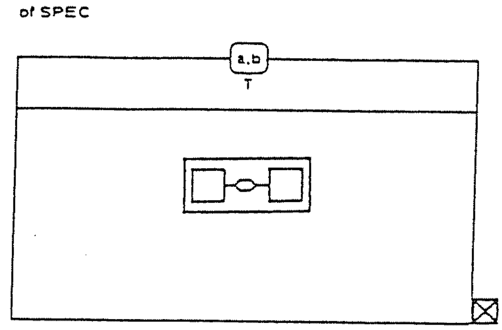


Figure 2.16 The process T with folded behaviour

The definition of process T where the fully expanded behaviour is replaced by the icon for parallel composition is shown in Figure 2.16.

3. Overview of the definition of the G-LOTOS syntax

In this section we sketch the key ideas on which our formal definition of the G-LOTOS syntax is based. The complete formal definition of a subset of the syntax can be found in [BL89a], while alternative techniques for the definition of G-LOTOS are discussed in [BL89b]. The three main ingredients of the formal definition are the abstract syntax of LOTOS, called A-LOTOS, a picture definition language, called Pictor, and a relation between A-LOTOS terms and Pictor terms, called 'representation'.

3.1. A-LOTOS

A-LOTOS is the language of *Abstract Syntax* terms for LOTOS. An A-LOTOS term describes a LOTOS construct abstracting form details such as the actual symbols used in any particular concrete (textual or graphical) representation, or operator associativity and precedence in textual representations. Formally, the grammar for A-LOTOS is defined in terms of *classes*, which correspond to the syntactic categories of the language, and n-ary *operators* (n≥0), which correspond to the constructs of the language. An example of the definition for a subset of the class *Behaviour* for Basic LOTOS (that is, LOTOS without data types and values) is given in Figure 3.1. A BNF-like meta-notation has been used: class names (nonterminals) have upper case initials, while lower case is used for operator names (terminals); alternatives are separated by '|' while '(' and ')' are considered as terminal symbols.

```
Behaviour ::=
|  choice(Behaviour, Behaviour)
|  obs_action(Id, Behaviour)
|  stop
```

Figure 3.1 - A fragment of A-LOTOS syntax

In the example of Figure 3.1 *stop* is the only nullary operator while *choice* and *obs-action* are both binary operators (*Id* stands for the lexical category of identifiers, not defined here). For example, the abstract syntax term corresponding to the standard LOTOS expression 'a; b; stop [] stop' is:

choice(obs-action(id(a),obs-action(id(b),stop), stop).

Note that, for convenience, we include actual identifiers such as 'a' and 'b' in the abstract syntax terms, although, stricly speaking, they are not part of the abstract syntactic structure of a LOTOS specification.

3.2 Pictor

In this section we give an overview of Pictor, a language for describing graphic objects. It is worth noting that Pictor is not intended to be a general purpose picture description language, and its operators are not claimed to satisfy any orthogonality requirement. In fact, a potential improvement of the formal definition of G-LOTOS could possibly derive from the adoption of some existing, perhaphs standardized picture definition language. The syntax of Pictor is quite simple. A fragment of the Pictor grammar is given in Figure 3.2, where the same notational conventions adopted for the A-LOTOS grammar apply.

```
Pictor-term::=
    crossed_circle
|   "Text"      (* any string of text in quotes *)
|   box
|   dash_box
|   oval
|   arrow
|   inside_c.c(Pictor_term, Pictor_term)
|   same_outrect(Pictor_term, Pictor_term)
|   aside_sw.nw_se.ne(Pictor_term, Pictor_term)
|   aside_ne.nw_se.sw(Pictor_term, Pictor_term)
|   connection(Pictor_term, Pictor_term)
```

Figure 3.2 - A fragment of the Pictor syntax

In the following, for notational convenience, indentation will be used instead of parenthesis for Pictor terms, so that

```
operator
   Term1
   Term2
```

will stand for 'operator(Term1,Term2)'.

Pictor terms are meant to describe pictures in an abstract way. This means that details such as actual sizes or orientations of the picture elements are deliberately not captured by Pictor terms. For example, the Pictor term shown in Figure 3.3, is a description of the picture in Figure 3.4, but it also describes all the variants obtained by stratching and rotating it in various ways.

```
same_outrect
  aside_sw.nw_se.ne
    inside-c.c
      connection
        inside_c.c
          "a"
          oval
        connection
          arrow
          connection
            inside_c.c
              "b"
              oval
            connection
              arrow
              crossed_circle
      dash-box
    inside_c.c
      crossed_circle
      dash_box
  box
```

Figure 3.3 - A Pictor term

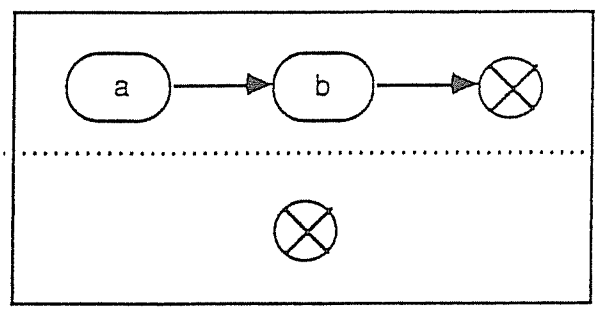


Figure 3.4 - A picture (instance)

In order to formalize the interpretation of Pictor terms as actual pictures, we need to characterize pictures as formal objects. We do this by the following definitions.

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

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

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

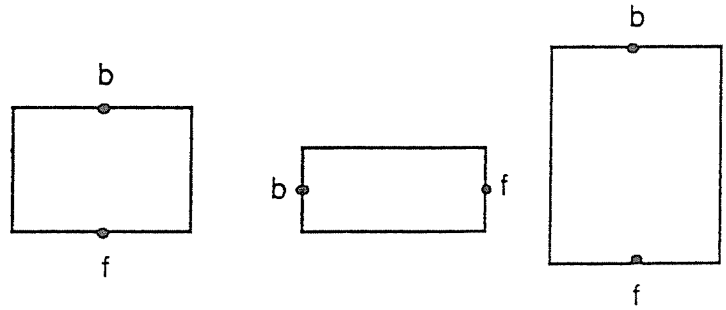
A *picture* is a set of picture instances. We will build complex pictures in terms of *basic pictures* Among basic pictures we distinguish between *constant-size* and *variable-size* pictures.

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

A *variable-size basic picture* is a basic picture which is not a constant-size one; typically, the instances of a variable-size picture are obtained by freely stretching along the cartesian axes a unique picture instance.

One instance of the constant-size basic picture *crossed-circle*, and three instances of the basic picture *box*, of variable size, are shown in Figure 3.5 (a) and (b).

(a) crossed-circle



(b) box

Figure 3.5 - Constant and variable-size basic pictures.

In order to generate elementary graphic objects, we assume the existence of a number of primitive *basic picture predicates*, named as the basic pictures, which may be satisfied by picture instances. Thus, picture P can be defined as the set of picture instances which satisfy the predicate P associated to it:

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

Predicates are useful since they allow one to characterize pictures at a convenient level of abstraction: by using predicates, we do not care about the *absolute* positions of picture instances on the plane, or about their *actual* sizes.

Figure 3.6 lists all the *basic picture predicates* related to our working example. In the figure, each predicate is associated with some picture instances which satisfy it. The distinguished points Back and Front are put in evidence, and labelled 'b' and 'f'; S is the actual set of points of the picture instance. Clearly one could take the burden of providing an equational characterization of the basic picture predicates; however, given the low level of complexity of the involved pictures, the assumption of *primitive* predicates, introduced by pictorial example, has seemed sufficiently unambiguous.

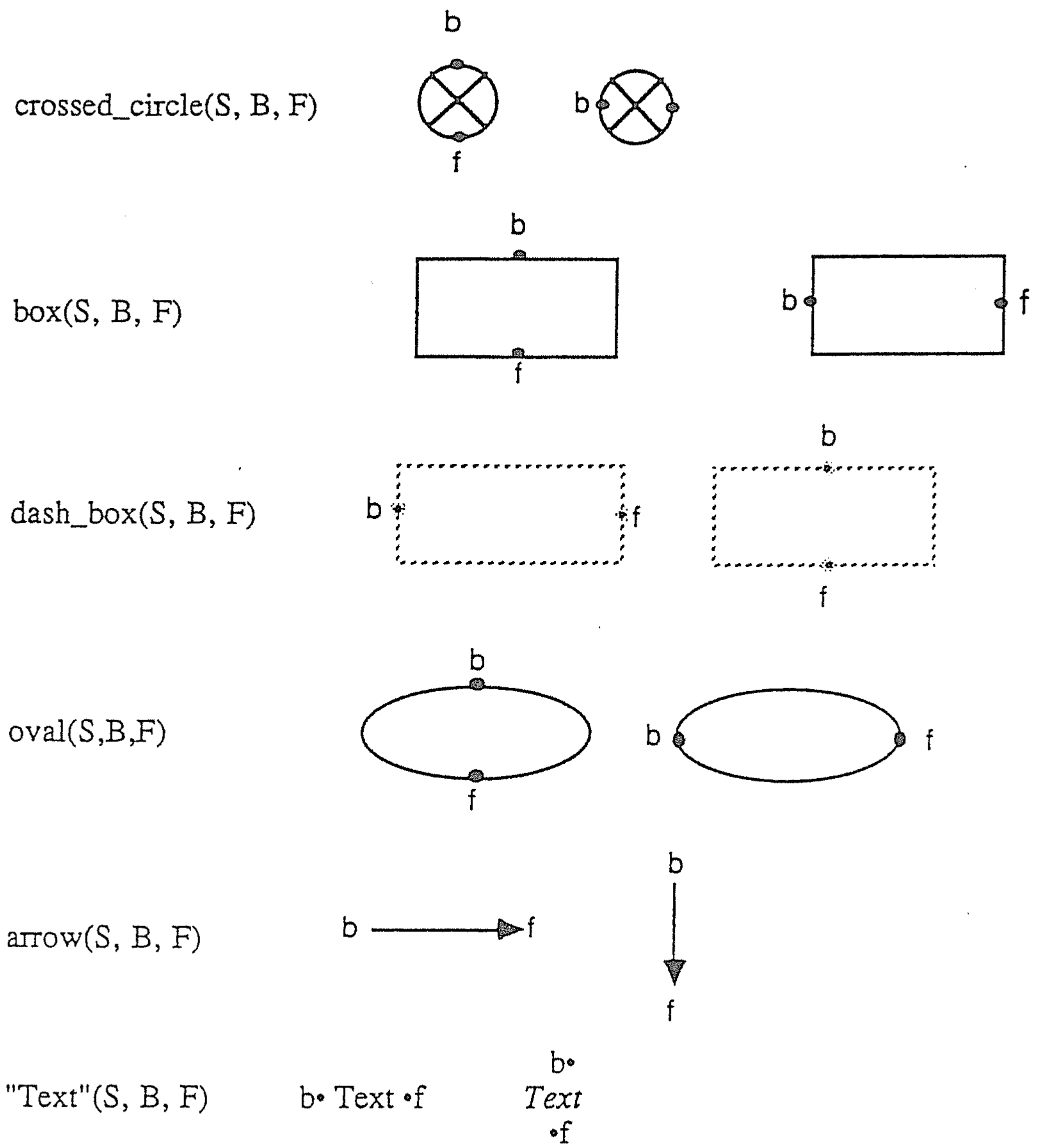


Figure 3.6 - Basic picture predicates

In order to interpret the remaining, non-nullary operators of Pictor (e.g., 'inside_c.c(Pictor_term, Pictor_term)'), we need to be able to combine elementary graphical objects into complex ones, and to impose constraints on their *relative* positions and *relative* sizes. We do this by using further *primitive* predicates. Together with the basic picture predicates listed above, these predicates form the *kernel* our graphic specification model. Before introducing them, we need to introduce the notions of *reference rectangle* and *reference point*.

In order to handle pictures conveniently, each one of them is associated with two invisible *reference rectangles* and eighteen invisible *reference points* (besides the Front and Back distinguished points). The two reference rectangles are identified by the two *reference rectangle identifiers* 'outrect' and 'inrect'.

Figure 3.7.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).

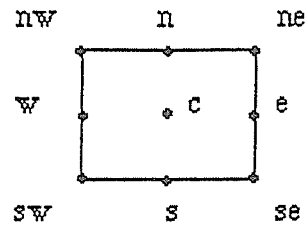


Figure 3.7 : Reference rectangles and points.

We are ready for introducing the required primitive predicates. Each predicate is introduced in informal, declarative style: the predicate name is followed by the list of arguments; then, a short text is provided, in *italics*, which describes the conditions by which the predicate is satisfied.

The primitive predicate 'ref_point' identifies any subset of the reference points of either reference rectangle ('inrect' or 'outrect') associated to a picture :

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'.

Of course the *outrect* rectangle is defined also for text-pictures.

Further primitive predicates are useful for building complex pictures: they are listed below.

union ([S1, ..., Sn], S) *if*
the set of points S is the union of the sets of points S1, ..., Sn.

disjoint ([Picture1, ..., Picturen]) *if*
the intersection of the areas delimited by the 'outrect' rectangles of pictures Picture1, ..., Picturen is empty, or is of null area.

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

common_points (Point_list1, Point_list2) *if*
there exists an element common to the two lists Point_list1 and Point_list2.

Based on the primitive predicates introduced above, we can formalize the interpretation of Pictor terms as actual pictures. This is achieved via the relation *fig*, which associates Pictor terms with actual graphical objects, that is, elements of the set *Pictures*:

$$\text{fig} \subseteq \text{Pictor} \times \text{Pictures}$$

Relation *fig* is defined below in declarative style; one clause is provided for each one of the Pictor operators listed in Figure 3.2. For example, the first clause below reads as follows: Pictor term 'crossed_circle' is in relation 'fig' with picture (S, B, F) (that is, the latter is a pictorial interpretation of the former) if (S, B, F) satisfies the predicate **crossed_circle.**

```
fig(crossed_circle, (S, B, F))    if   crossed_circle(S, B, F).
fig("Text", (S, B, F))            if   "text" (S,B,F).
fig(box, (S, B, F))               if   box(S, B,F).
fig(dash_box, (S, B, F))          if   dash_box(S, B,F).
fig(oval, (S, B, F))              if   oval(S, B,F).
fig(arrow, (S, B, F))             if   arrow(S, B,F).
```

Clearly the definition of relation 'fig' is trivial for nullary Pictor operators; it becomes more interesting for non-nullary operators, as shown below. Each clause of the definition is followed by a short explanation in *italics*.

```
fig(inside_c.c(X, Y), (S, B, F)) if
        fig(X, (XS, XB, XF)),
        fig(Y, (YS, B, F)),
        ref_rect(XS, outrect, Xoutrect),
        ref_rect(YS, inrect, Yinrect),
        rect_contains(Yinrect, Xoutrect),
        ref_points(XS, outrect, [c], Xc),
        ref_points(YS, inrect, [c],Yc),
        common_points(Xc, Yc),
        union([XS, YS], S).
```

In order for (S,B,F) to be a picture associated to Pictor term 'inside_c.c(X,Y)', it is required that S be the union of the pictures associated to X and Y , that the outrect of the first one be contained in the inrect of the second one and that such rectangles be centered. The Back and Front of the new picture are defined to be those of the external component .

```
fig( same_outrect(X,Y), (S, B, F)) if
        fig(X, (XS, XB, XF)),
        fig(Y, (YS, B, F)),
        ref_points(XS, outrect, [ne, sw],[Xne, Xsw]),
        ref_points(YS, inrect, [ne, sw],[Yne, Ysw]),
        common_points([Xne], [Yne]),
        common_points([Xsw], [Ysw]),
        union([XS, YS], S).
```

In order for (S, B, F) to be a picture associated to Pictor term 'same_outrect(X,Y)', it is required that S be the union of the pictures associated to X and Y, and that the outrect of the first one be equal to the inrect of the second one.

```
fig(aside_sw.nw_se.ne(X,Y), (S, B, F)) if
        fig(X, (XS, B, XF)),
        fig(Y, (YS, YB, F)),
```

```
        ref_points(XS, outrect, [sw], Xsw),
        ref_points(XS, outrect, [se], Xse),
        ref_points(YS, outrect, [nw], Ynw),
        ref_points(YS, outrect, [ne], Yne),
        common_points(Xsw, Ynw),
        common_points(Xse, Yne),
        union([XS, YS], S).
```

In order for (S,B,F) to be a picture associated to Pictor term 'aside_sw.nw_se.ne(X,Y)', it is required that S be the union of the pictures associated to X and Y and that the 'sw', respectively 'se' point of the first one coincide with the 'nw', respectively 'ne' point of the second one.

```
fig(aside_ne.nw_se.sw(X,Y), (S, B, F)) if
        fig(X, (XS, B, XF)),
        fig(Y, (YS, YB, F)),
        ref_points(XS, outrect, [ne], Xne),
        ref_points(XS, outrect, [se], Xse),
        ref_points(YS, outrect, [nw], Ynw),
        ref_points(YS, outrect, [sw], Ysw),
        common_points(Xne, Ynw),
        common_points(Xse, Ysw),
        union([XS, YS], S).

fig(connection(X, Y), (S, B, F)) if
        fig(X, (XS, B, C)),
        fig(Y, (YS, C, F)),
        ref_rect(XS, outrect, Xoutrect),
        ref_rect(YS, outrect, Youtrect),
        disjoint([Xoutrect, Youtrect]),
        union([XS, YS], S).
```

In order for (S, B, F) to be a picture associated to Pictor term 'connection(X,Y)', it is required that S be the union of the pictures associated to X and Y and that the front point of the first one coincide with the back point of the second one. Moreover, it is required that the outrects of such pictures do not overlap.

3.3 Mapping A-LOTOS <--> Pictor

The identification of all and only those concrete representations that can be associated with the abstract syntactic structure of a given LOTOS construct is given here by defining the relation *representation* between A-LOTOS and Pictor:

$$\text{representation} \subseteq \text{A-LOTOS} \times \text{Pictor}$$

Relation *representation* is defined in terms of two relations *pict* and *text*, which are first introduced informally below. They are useful for adequately mixing textual and pictorial forms of G-LOTOS.

pict (A-LOTOS-term, Pictor-term) *if*
Pictor-term denotes a legal concrete representation of A-LOTOS-term; the representation is 'graphical' in the sense that at least the outermost operator of A-LOTOS-term is represented pictorially.

text (A-LOTOS-term, Pictor-term) *if*
Pictor-term denotes a legal concrete representation of A-LOTOS-term; the representation is completely textual.

representation (A-LOTOS-term, Pictor-term) *if*
text (A-LOTOS-term, Pictor-term).

representation (A-LOTOS-term, Pictor-term) *if*
pict (A-LOTOS-term, Path, Pictor-term).

Pictor-term denotes a legal concrete representation of A-LOTOS-term; the representation may be either graphical or textual.

The definition of the *representation*, *pict* and *text* predicates above expresses a specific criterion that we have adopted for defining the legal G-LOTOS representations. A pictorial representation of a construct may include the textual representation of some subconstruct; thus a clause of the *pict* predicate may invoke the *repr*, the *pict* and the *text* predicates. On the contrary, a textual representation will only be entirely textual, hence a clause of predicate *text* may only invoke predicate *text* itself.

In the following subsections we define in detail the *pict* and *text* predicates (thus the *representation* predicate) for the subclass of *Behaviour* defined in Figure 3.1, that is, for the three LOTOS operators *choice*, obs_*action (prefix)*, and *stop*.

Formal definition of predicate *pict*.

$$\text{pict} \subseteq \text{A-LOTOS} \times \text{Pictor}$$

The most simple clause for relation *pict* is the one which associates the *stop* A-LOTOS (and LOTOS) operator to the *crossed_circle* Pictor operator:

pict (stop, crossed_circle).

Two clauses are needed for building the graphical representation of the LOTOS *choice* operator, that is, the box divided into two sub-boxes by a dotted line, since the two parts can be arranged one on top of the other, as in Figure 3.4, or horizontally, side by side. The two cases are handled, respectively, by the second clause and first clause below.

```
pict (  choice(Behaviour1, Behaviour2),
        same_outrect
            aside_ne.nw_se.sw
                inside-c.c
                    P_Behaviour1
                    dash-box
                inside_c.c
```

```
                P_Behaviour2
                dash_box
        box
    ) if
    repr (Behaviour1, P_Behaviour1),
    repr (Behaviour2, P_Behaviour2).

pict (  choice(Behaviour1, Behaviour2),
        same_outrect
          aside_sw.nw_se.ne
            inside-c.c
                P_Behaviour1
                dash-box
            inside_c.c
                P_Behaviour2
                dash_box
          box
    ) if
    repr (Behaviour1, P_Behaviour1),
    repr (Behaviour2, P_Behaviour2).
```

The pictorial representation of an action prefix construct requires an extra box in case the representation of the behaviour following the action has not a box shape. For solving this problem we have introduced an auxiliary predicate:

```
prefixable(Behaviour, P_ Behaviour) if
    pict(Behaviour, P_Behaviour).

prefixable(Behaviour,
        inside_c.c
            P_Behaviour
            box
    ) if
    repr(Behaviour, P_Behaviour).
```

By using the newly introduced predicate we can safely define the pictorial representation of the *obs_action (prefix)* construct:

```
pict (  obs_action(Gate, Behaviour),
        connection
            inside_c.c
                P_ Gate
                oval
            connection
                arrow
                P_ Behaviour
    ) if
    text(Gate, P_Gate),
    prefixable(Behaviour, P_ Behaviour).
```

Predicate *text*

The definition of the *text* predicate, that is, of the relation between A-LOTOS terms and their textual LOTOS representations does not deserve detailed discussion. A sketch of such a definition can be found in [BLP88]. The only aspect of a certain interest is concerned with associativity and precedence rules for (textual LOTOS) binary operators. For instance, any term *Text* which is the concatenation of *Text1* , the symbol *[]* and *Text2*, is a textual representation of the Abstract LOTOS term *choice(Bex1, Bex2)*, provided that *Text1* and *Text2* are textual representations of, respectively, *Bex1* and *Bex2*. However some asymmetry appears in handling the right hand side and the left hand side of a choice. For example, if *Bex1* is a choice then its textual representation in the above choice context is any valid textual representation for *Bex1*, provided it be enclosed in parenthesis.

text (A_LOTOS_term, P_Pictor_term) if

Pictor_term is a textual representation of A_LOTOS_term according to the textual syntax defined in IS 8807.

In conclusion, G-LOTOS is formally defined as follows:

G-LOTOS = {S | S ∈ Pictures, exist p ∈ Pictor, t ∈ A-LOTOS, representation(t,p), fig(p,S)}

From the definition of the *representation* relation it turns out that the A-LOTOS term

choice(obs-action(id(a),obs-action(id(b),stop), stop)

used in a previous example can be associated to the Pictor term shown if Figure 3.3, which, in turn, will be interpreted as the picture shown in Figure 3.4. On the other hand, by choosing the 'textual branch' of the *representation* relation, other Pictor terms can be associated to the above A-LOTOS term, where a mixture of textual and graphical representations is present. One of them is given in Figure 3.8 (a), and its interpretation is 'a;b;stop[]stop'; another one is given in Figure 3.8 (b), and its interpretation is shown in Figure 3.9.

```
connection
  "a"
  connection
      ";"
      connection
          "b"
          connection
              ";"
              connection
                  "stop"
                  connection
                      "[]"
                      "stop"
```

(a)

```
same_outrect
    aside_sw.nw_se.ne
        inside_c.c
            connection
                "a"
                connection
                    ";"
                    connection
                        "b"
                        connection
                            ";"
                            "stop"
            dash-box
        inside_c.c
            crossed_circle
            dash_box
    box
```

(b)

Figure 3.8 - Sample Pictor terms

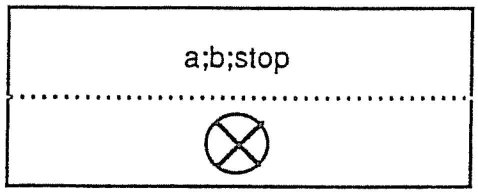


Figure 3.9 - A sample picture

4. Specification of a G-LOTOS Editor in LOGGIE

In this section, we describe the meta-notation for specification of editors in LOGGIE. Our illustration is based on the subset of G-LOTOS already adopted in the previous sections. It must be observed that the LOGGIE specification of the editor cannot be *immediately* derived from the definition of G-LOTOS given in Section 3, although the abstract syntax is the central element in both cases; one should more correctly view the former as an *implementation* of the latter. The meta-notation in LOGGIE is described in section 4.1. Basic pictures are introduced in section 4.2 and are defined in terms of graphical data types. In section 4.3 the mapping from abstract syntax terms to graphics is defined. In sections 4.4 to 4.7 this mapping is further defined and examplified in the case of our working example.

4.1 Meta-notation

Editor specifications are defined in the meta-notation of LOGGIE. The meta-notation is used for formulating the abstract syntax with its attributes and garlands. The abstract syntax together with the attributes and attribute equations form an *attribute grammar* (AG), while garlands are links between nodes of abstract syntax trees. Garlands define an extension to the syntax; we may derive abstract syntax 'graphs' instead of simply 'trees'. One of the benefits of garlands is the ability to express non-local attribute dependencies [DRZ85].

Production rules of the abstract syntax are entered in an extended BNF graph form, called Graphical Extended BNF. GEBNF consists of *nodes* and *syntax operators* connected by links; an example is provided in Figure 4.1, where syntax operators are surrounded by thick rectangles. The extension of BNF consists of three list operators: '*', '+' and '|*' (List, List of at least one element, List of mixed types). The list operators are not used in our working example, but may be used whenever lists are required in a language, e.g. a list of identifiers.

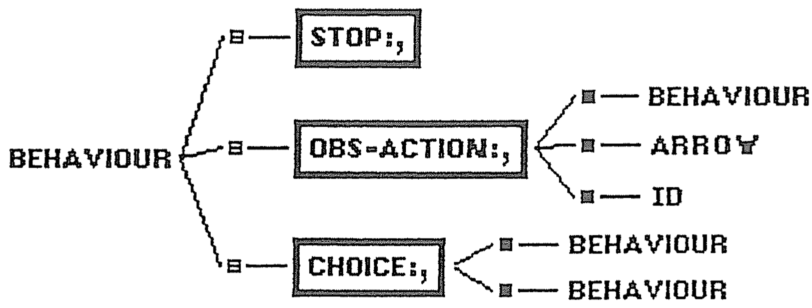


Figure 4.1 Abstract Syntax in graph form

The classes of the abstract syntax, mentioned in section 3, are mapped to GEBNF nodes. Abstract syntax n-ary operators and constants are mapped to GEBNF syntax operators. As an example, Figure 4.1 shows the GEBNF version of the abstract syntax given in Figure 3.1. The class *Arrow* has been added to the abstract syntax. This had to be done since each basic picture must be associated with a symbol of the grammar, and in LOGGIE a connector is a basic picture. Such an extension of the abstract syntax is a refinement of the specification defined in Section 3.

Derivation of terms of the abstract syntax is accomplished in a top-downcommon to many language-based systems. The abstract syntax ensures that only well-formed terms will be generated. The main operation for manipulating terms is *Replace* which replaces one well-formed term with another. *Add* adds a new member to a list and is consequently only defined for terms associated with list operators ('*', '+' and '|*'), and *Remove* is defined for operands of such lists.

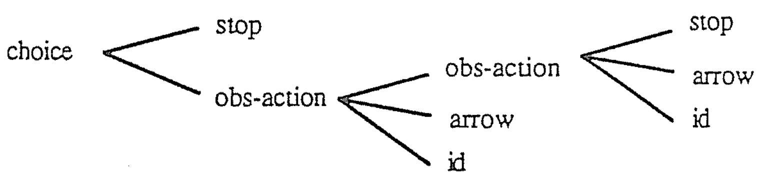


Figure 4.2 A Derivation Tree

A *derivation tree* is associated to each derived term whose nodes are called *derivation nodes.* As an example, Figure 4.2 gives the derivation tree of the term: 'choice(obs-action(id,arrow,obs-action(id,arrow,stop), stop)' as a derivation tree. This is a term of the abstract syntax of Figure 4.1.

In an AG, *attribute equations* are associated to the context free productions. Each attribute equation defines the value of one attribute belonging to one grammar symbol occuring in the production. An AG defines a number of attribute instances for each derivation node. A derivation node together with its attribute instances (attribute values) is called a *semantic node*. The derivation tree together with the attribute instances of all derivation nodes of the tree is called a *semantic tree*. A semantic tree is a tree of semantic nodes.

4.2 LOGGIE Basic Pictures

As is the case for the PICTOR language, LOGGIE Pictures are either basic or composite. Basic pictures are defined as graphical data types in the graphics system, SPL, while composite pictures are compositions of such pictures. Basic pictures are classified according to their nature, with different commands and data fields. The classification of basic pictures together with the pictures used in this section are shown in Figure 4.3.

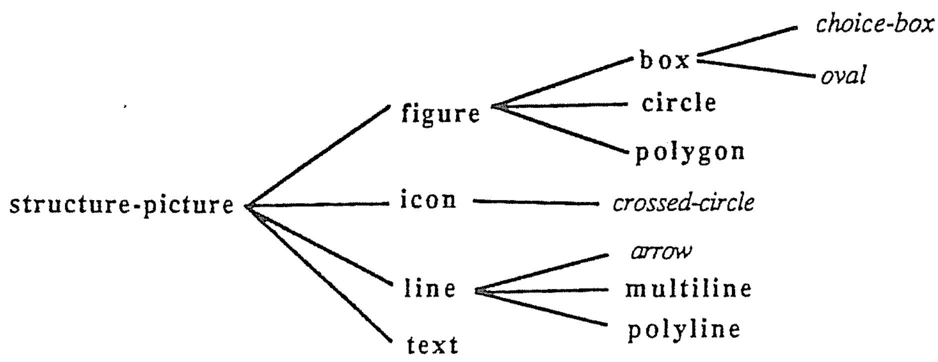


Figure 4.3 Classification of Basic Pictures

The general classification of basic pictures is into icons, figures, lines and text. *Icons* are fixed size graphical constants, comparable with the *constant-size* pictures of Section 3. *Figures*, on the other hand, may be resized and are comparable with the *variable-sized* pictures. Typically, figures have contents, i.e. pictures that are constrained to reside within the region of the figure. Some of the specializations of figure is *box*, *circle* and *polygon*. *Lines* are stretchable pictures with endpoints that may be attached to other pictures; when the location of the connected pictures are changed, the line is stretched accordingly. Among the specializations of line are *polyline* (a broken line) and *multiline* (a line with more than two endpoints). Finally, *text* is just a concatenation of characters.

In the following sections, we will show how the basic pictures of SPL may be composed into composite pictures.

Some of the fields of a basic picture are:

image	the internal image description, i.e. the way a basic picture is presented.
in & out-points	attachment points for connections.
input definitions	default interaction definitions, i.e. the way a user may interact with a basic picture.
marking	how a picture is marked, i.e. highlighting, shading.

Some of the Pictor terms of section 3 may now be implemented as LOGGIE basic pictures. For instance, *crossed-circle* (Figure 3.1) is an example of an icon, a graphical constant with fixed size. The *oval* and *box* terms (Figure 3.2) are examples of figures. The *arrow* term (Figure 3.2) is a specialization of *line*, a stretchable connection between points. Note however, that the basic pictures of LOGGIE implementing the pictor terms includes more information, such as user interaction and marking.

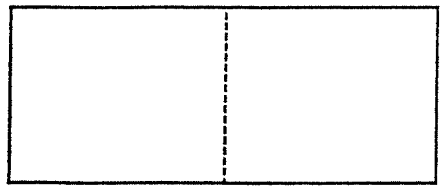


Figure 4.4 A Choice-box

For our working example a *choice-box* (Figure 4.4) is needed. The choice-box is a specialization of the *box* figure, consisting of a box with a dashed vertical line. Choice-box is an implementation of the Pictor term which is related by the *pict* relation (Section 3) to the abstract LOTOS term 'choice(X,Y)'.

The 'back' and 'front' points of Pictor terms are implemented by 'in & out-points' of basic pictures. 'Input definitions' are added to basic pictures in order to enable user interaction with a picture. The input definitions include specification of mouse and keyboard interaction (see Section 4.6).

4.3 Mapping from Semantic Trees to Graphics

Since the concrete syntax of a graphical language is much too complex to be expressed by pretty-printing rules, we need more powerful methods. This is done in LOGGIE by separating the graphical system from the derived structures, and by defining a service interface. Relations, interactions and the picture compositions are handled in abstract terms, and reference to basic pictures is symbolical. There is no need to specify coordinates and other details, since these are dealt with by the graphical system.

The mapping to graphics is controlled by attributes. Values of attributes control how a semantic tree is presented in the graphical system, and thereby maps a semantic tree on basic pictures. The mapping from a semantic tree may be divided into separate mappings for each node in the tree. The complete mapping for a semantic tree is the composition of the mappings of each semantic node. The mapping from nodes to graphics have the following parts:

basic pictures	basic picture elements
composition	complex pictures
interaction	user interaction with pictures
marking	when and how pictures are marked

Each of the four parts of the mapping is specified by a class of attributes. The remaining sections describe each part of the mapping with associated attribute classes, examplified by our working example. In each section, an example attribute grammar will be presented to illustrate the mapping. Note that these attribute grammars can be composed into one attribute grammar defining the whole mapping, although, for the sake of simplicity, such composition is omitted.

4.4 Mapping from Semantic Trees to Basic Pictures

Basic pictures are associated with derivation nodes and attribute values. In fact, many basic pictures can be associated with the same derivation node, each one displaying one *aspect* of the node. Furthermore, each picture may have several *views*. The association of basic pictures to derivation nodes is accomplished by defining a *picture* attribute in the grammar. The value of *picture* is a list of basic picture names, each picture corresponding to one aspect. Figure 4.4 shows the abstract syntax of Figure 3.1 expanded with attribute equations for *picture*. Normally, the aspects would also have been included, but they have been omitted for the sake of clarity.

```
Behaviour      ::=
        choice
                (Behaviour, Behaviour)
                {Behaviour1.picture = (choice-box)}
|       obs-action
                (Id, Arrow, Behaviour )
                {Behaviour1.picture = (),
                Id.picture = (oval),
                Arrow.picture = (arrow)}
|       stop
                {Behaviour.picture = (stop)}
```

Figure 4.5 Associating basic pictures to grammar symbols.

Figure 4.5 is an example of an attribute grammar specification and introduces the attribute *picture*. At this point a few words about the notation are appropriate. Attribute a of grammar symbol A is referenced to as $A.a$. An *attribute equation* defines the value of one attribute and is associated with a rule of the grammar. The format of such an equation is $A.a = expr$, where $A.a$ is an attribute and *expr* is a function applied to other attributes. Subscripts are used to identify different instances of the same grammatical symbols. As an example, $Behaviour_2$ indicates the *second* occurence of the *Behaviour* symbol in a given rule.

It is also possible to associate attributes to basic pictures. This feature is however limited to basic pictures of type *text*. The textual presentation is useful if an attribute contains relevant textual information, such as a title, an error message or a translation, as it happens when some code is generated.

4.5 Composite Pictures

Composite pictures are built by combining basic pictures. A composite picture can be composed by defining *hierarchy*, *spatial relations* and *connections* between basic pictures. The abstract syntax tree defines the default hierarchy of pictures, i.e. which pictures are sub pictures of others. In the case of Figure 4.2, the picture associated with 'choice' (*choice-box*) has two sub pictures: the pictures of 'stop' and 'obs-action'.

The default hierarchy given by the abstract syntax tree can be overruled by the *subs* attribute. The value of *subs* for a derivation node is a list of derivation nodes and attribute names defining the sub pictures of the composite picture defined for that node. In this way, we may compose pictures in a different way than the abstract syntax imposes. However, in our working example, the abstract syntax suffices to define the composition of pictures.

In Section 4.4 we showed how aspects can be defined in order to associate different basic pictures to derivation nodes. Aspects can also be used to compose basic pictures in various ways. In the example of Figure 4.2 we could, for one aspect of 'choice', choose to display a textual aspect of 'stop' together with the graphical aspects of the other nodes.

So far we have only showed how composite pictures are composed by combining basic pictures in a tree formed hierarchy, the next step is to add constraints on their layout. This is done by adding spatial relations holding between pictures and their immediate parents in a hierarchy. Some examples of such relations needed for our example are given below:

inside (basic-picture$_1$, basic-picture$_2$) if
basic-picture$_1$ is inside basic-picture$_2$.
inside-left (basic-picture$_1$, basic-picture$_2$) if
basic-picture$_1$ is inside the left half of basic-picture$_2$.
inside-right (basic-picture$_1$, basic-picture$_2$) if
basic-picture$_1$ is inside the right part of basic-picture$_2$.
fix$_{ij}$ (basic-picture$_1$, basic-picture$_2$) if
basic-picture$_1$ is attached by the point given by i to the point given by j of basic-picture$_2$

Here i and j are pairs of rational numbers in the range [0..1] indicating relative positions in the relative coordinate system of each basic picture. For all relations, the first operand (*basic-picture1*) is a child of the second (*basic-picture2*) in some hierarchy.

```
Behaviour      ::=
        choice
                (Behaviour, Behaviour)
                {Behaviour2.relation = inside-left,
                 Behaviour3.relation = inside-right}
|       obs-action
                (Id, Arrow, Behaviour )
                {Id.relation = inside),
                 Behaviour2.relation = inside) }
|       stop
```

Figure 4.6 Defining relations

The *relation* attribute implements the spatial relation of pictures. The operands of the relation are implicit from the hierarchy and are not stated in the attribute value. The first operand of a relation is always the node that the attribute belongs to. The second operand is the parent of that node for some hierarchy. Figure 4.6 gives an example of the relations needed for our working example. In the first rule (*choice*), the pictures associated to the two *Behaviours* should be placed in the left and right half of the picture associated to *choice*. In the second rule (*obs-action*), the pictures of *Id* and *Behaviour* should be placed inside the picture associated to *obs-action*. Note that the relations are stated to hold between the *basic pictures* associated to the derivation nodes given by the *picture* attribute (see Section 4.4).

An essential ingredience in most graphical languages is the concept of *connectors*. LOGGIE supports connectors by the basic picture type *line* (see Section 4.2); a picture which may be connected to other pictures. The connections of a line picture are defined by the value of the *connection* attribute. The value of *connection* contains a list of garlands and/or attribute names which, in their turn, specify the objects whose pictures shall be connected. The only connector in our example is the *arrow* connection of the observable action prefix.

```
Behaviour       ::=
        choice
                (Behaviour, Behaviour)
                {Behaviour1.to-object = Behaviour1}
|       obs-action
                (Id, Arrow, Behaviour )
                {Arrow.connection = (from to),
                 Arrow.from = Id,
                 Arrow.to = Behaviour2,
                 Behaviour1.to-object = Id }
|       stop
                {Behaviour.to-object = Behaviour}
```

Figure 4.7 Defining the action prefix connector

Figure 4.7 introduces four attributes: *connection*, *from*, *to* and *to-object*. The *connection* attribute of *obs-action* lists the two attributes *from* and *to* (of *Arrow*), which specify the derivation nodes connected by *Arrow*. The value of *from* is *Id*, i.e. the picture associated with *Id*, while the value of *to* is context-dependant in the form of the synthesized attribute *to-object*. *To-object* returns the $Behaviour_2$ object in the case that it is a *stop* or a *choice*. If $Behaviour_2$ is an *obs-action*, then *to-object* returns the *Id* of that rule. The in- and out-points of the basic pictures of the connected nodes will determine the actual geometric connection points of the arrow.

4.6 User Interaction

Up to now, we have only regarded the presentation of a derived structure in terms of basic and composite pictures. Since we are dealing with interactive environments, we also want to define how a user may *interact* with these pictures. One of the fields of a basic picture is named *input definitions* (see Section 4.2). The input definitions define interaction in terms of applicable operations on a picture, and how to access theses operations by commands and menus. Some operations are 'general purpose' in the sense that they do not affect any attribute values or derivation trees; such as *shape*, *move* and *print*. The 'general purpose' operators may change values of the basic pictures or access the environment (printer, network etc), but they do not modify the underlying sentence of a language. Other operations involve modifying the derivation tree and its attributes, such as *replace*, *add*and *remove*.

In order to make the set of operations context-dependant, LOGGIE offers a mechanism of altering the input definitions with the help of attributes. This section gives an example of user interaction and how this is specified in LOGGIE.

In our example, there is a *Standard menu* associated to all behaviours, with the following items:

cut cutting a subtree
copy copying a subtree
box the behaviour is enclosed within a box
text/graphics the representation is changed between graphical and textual

In Figure 4.8 a user has interacted with a *stop* behaviour by pointing to it with the mouse and clicking. The standard menu has appeared, and the user may choose one of the items in the menu.

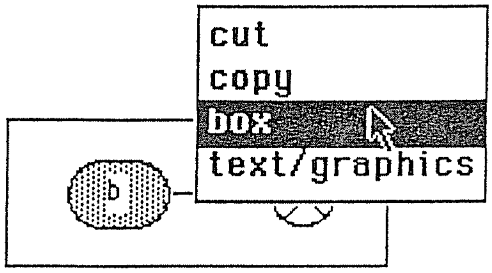


Figure 4.8 Selecting the standard menu for *stop*

For some behaviours, the standard menu is extended with additional items. For instance, the menu of 'obs-action' is extended with *edit gate*, a function which allows to change the *gate* attribute of 'Id'.

The ways in which the end user will interact with the editor are defined partly when the basic pictures are defined, partly by attributes. The attribute *menus* of a derivation node specifies how to interact with a picture in order to visualize menus, and which attributes contain the menu descriptions. The value is a list of pairs:

<input-definition, attribute >

where *input-definition* specifies how the menu is accessed and *attribute* is a name of an attribute containing a menu description. In other words, *menus* describes how menus are accessed and where to find their definitions.

```
Behaviour.menus = (<middle  stdmenu>)
Behaviour        ::=
        choice
                (Behaviour, Behaviour)
                {Behaviour1.stdmenu = (cut paste box text/graphics)}
|       obs-action
                (Id, Arrow, Behaviour )
                {Behaviour1.stdmenu =  (cut paste box text/graphics editgate)}
|       stop
                {Behaviour.stdmenu = (cut paste box text/graphics)}
```

Figure 4.9 The specification of the Standard menu for behaviours.

In Figure 4.9 the *menus* attribute specifies that the middle mouse button is used to access the standard menu for behaviours. It also specifies which attribute contains the menu items, namely *stdmenu*. Note that the equation for *menus* is stated by itself because of its independence of grammar rules.

4.7 Marking

A basic picture may be marked by highlighting, shading, markers etc. This is useful if we want to notify a user of some temporary information, such as a warning or an error. Marking may also be used in cases of animation. In this section we will use a static semantic condition to illustrate the use of markers in an editor.

Being based on attribute grammars, LOGGIE supports static semantics. For example, let us define the following static semantic rule:

The gate of the action prefix must be defined

The gate is implemented as the *gate* attribute of Id. Suppose we have an inherited attribute, *gates*, containing the list of formal gates (actually this attribute would be defined as the formal gate list of a process definition). In terms of an attribute equation, the rule can be stated as follows:

Id.gate **in** Behaviour$_1$.gates

Behaviour$_1$ is here the LHS of the obs-action production. Also, we want some means for indicating that the rule is not satisfied, maybe by shading the picture of Id, as in Figure 4.10

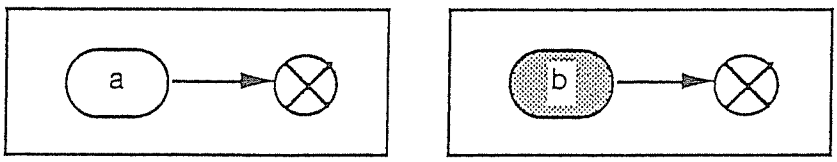


Figure 4.10 An example of error indication by marking objects; the gate 'b' on the right is undefined.

This is done by defining the *mark* attribute for a derivation node. Its value is a list of pairs <*marker*, *attribute*>, where *marker* is a marker defined in the graphic system and *attribute* is a name of an attribute controlling the marker; if the attribute is set, the picture of the node is marked.

```
Behaviour ::=
        obs-action
                (Id, Arrow, Behaviour )
                {Id.error = not (Id.gate in Behaviour1.gates),
                 Id.mark = (<shade error>)}
```

Figure 4.11 Specifying markers with attributes

In Figure 4.11, the *mark* attribute specifies that the basic picture associated with the derived instance of Id will be marked by *shade* if the *error* attribute is set.

5. Conclusions

We have presented the experimental prototyping of a graphical editor for the G-LOTOS syntax.

A natural feedback to be expected by such an experience is an evaluation of G-LOTOS as a visual language, that is, an assessment of the currently proposed G-LOTOS symbols and their combinations, with respect to the processes of building, browsing, interpretation of a LOTOS specification. However, we were not primarily driven by this aim, and we will refrain from discussions on aspects that depend to a considerable extent on personal taste. We found the G-LOTOS representations of the most fundamental LOTOS constructs (action prefix, sequential composition, parallel composition, choice, stop, exit, process definition and instantiation) implementable and useful. However, an experimental end-user test remains to be done. Perhaps not surprisingly, visual effectiveness is present in the intermediate phases of the editing process, when some graphical elements are still to be expanded; in fact, we suspect that the advantages of the graphical syntax over the textual one are more evident when incomplete specifications are considered.

Our prototype does not include the graphical representation of the 'disabling' construct, and of the unary operators of hiding ('hide'), local binding ('let'), multiple choice on gates or values ('choice'), and multiple parallel composition ('par'). Our impression is that the current version of the editor already supports the 'most interesting' part of G-LOTOS, and that inclusion of further graphic elements would not substantially enhance the readability of the produced specifications. An extension of the G-LOTOS editor would be useful for testing such opinion on an experimental basis.

Our main interests in prototyping the G-LOTOS editor were concentrated upon the methodology involved in the formalization of the graphical syntax, and in the use of advanced meta-tools for the rapid prototyping of the editor. In spite of the fact that the formal definition of G-LOTOS and the LOGGIE meta-tool come from completely independent research groups and efforts, nicely enough some matching between these two pieces of work has soon appeared, and has triggered the reported experiment. The key for the success of the latter has been the central role played by the abstract syntax in both cases; the actual prototype was developed by essentially one person in about three weeks.

Language-definition-directed meta-tools such as LOGGIE are meant to accept the formal definition of (some aspects of) a language, and provide software prototype tools for it in a highly automatized way. A well known tradeoff exists between the efficiency of the development phase and the efficiency of the developed tool. In this respect our G-LOTOS editor is not an exception. As far as the efficiency in the development is concerned, it is clear that a completely automatic implementation of the desired tool is out of question: some human participation is necessary, because some information needed for the implementation is either not expressed in the input formal definition, or is provided in a form which needs meta-tool-oriented translation. For instance, several implementation choices had to be made concerning the end-user interactions with the editor, the marking of graphic elements on the screen and visualization of static semantic errors; and (minor) modifications had to be made on the abstract syntax in order to reflect the tight coupling between abstract syntax tree nodes and graphical objects in LOGGIE.

As for the definition of basic pictures, and the expression of topologic relations among them, no substantial gap between the G-LOTOS definition and LOGGIE was found. However, some manual implementation was necessary at this level because the information provided, on one side, as an unparsing relation ('representation') between abstract LOTOS terms and Pictor terms, had to be re-distributed over abstract syntax trees, on the other side, in form of graphical node-attributes. We are not aware of the existence of language-definition-directed meta-tools so powerful as to directly accept the Prolog-like definition of the (graphical) unparsing relation as given in Section 3 (modulo the actual Pictor operators).

An interesting area of further research and experimentation with G-LOTOS and LOGGIE is concerned with the graphic representation of *dynamic semantic* aspects, such as displaying the action tree associated with a G-LOTOS specification, or the evolutionary stages of the behaviour it describes.

6. References

[BB87] T. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems, Vol. 14, No. 1, North-Holland, 1987.

[BHP89] B. Backlund, O. Hagsand, B. Pehrson, "Generation of Graphic Language-Based Environments", Research Report, Swedish Institute of Computer Science, SICS/R-89/8911, Stockholm, 1989.

[BL89a] T. Bolognesi, D. Latella, "A formal definition of a subset of G-LOTOS" - CNUCE Internal Report C89-23, 1989.

[BL89b] T. Bolognesi, D. Latella, "Techniques for the formal definition of the G-LOTOS syntax" - Proceedings of the 1989 IEEE Workshop on Visual Languages, IEEE Computer Society Press, 1989.

[BLP88] T. Bolognesi, D. Latella, A. Pisano, "Towards a graphic syntax for LOTOS" - Proceedings of EUTECO88 - Research into Networks and Distributed Applications, R. Speth editor, North-Holland, 1988.

[H84] C. J. Hogger, "Introduction to Logic Programming", Academic Press, 1984

[HN87] P. B. Henderson, D. Notkin, "Integrated Design and Programming Environments", IEEE Computer, Vol.20 Num.11, November 1987.

[ISO87] ISO/IEC JTC 1 / SC 21 / WG 1 N 541, "Proposals/ Approaches for a Graphical Representation of LOTOS" , December 1987.

[ISO88a] ISO/IEC JTC 1 / SC 21 N. 3253, "Proposal for a New Work Item to develop an addendum to ISO 8807 on graphical LOTOS", 1988.

[ISO88b] ISO/IEC JTC 1 / SC 21 / WG 1 N 675, "Status on work on G-LOTOS", Oct. 1988

[ISO88c] ISO/IEC JTC 1 / SC 21 / WG 1 N 687, "An enhancement of the methodologies proposed in document WG1 N 675 for the formal definition of G-LOTOS", Nov. 1988.

[ISO89] ISO - IS 8807, " Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", 1989.

[Knuth68] D.E. Knuth, "Semantics of context-free languages", Math. Syst. Theory Vol. 2, No.2, June 1968, pp.127-145

[RTD83] T. Reps, T. Teitelbaum, A. Demers, "Incremental context-dependent analysis for language-based editors" ACM Trans. on Programming Languages and Systems, Vol.5, No.3, July 1983, pp. 449-477.

[RT89] T. W. Reps, T. Teitelbaum, "The Synthesizer Generator - A system for constructing language-based editors", Springer-Verlag, New York 1989.

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

[XAIE] Interlisp-D Reference Manual, Vol. I - III, KOTO release, 1985. Xerox Lisp Release Notes, Medley Release, 1988.

[BCD88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascal, "Centaur: thesystem", Proc. of SIGSOFT'88, Third Annual Symposium on Software Development Environments, Boston, 1988.

[H86] Scott E. Hudson, Implementing a User Interface as a System of Attributes, SIGSOFT/SIGPLAN 1986.

[FZ89] Paul Franchi-Zannettacci, "Attribute Specifications for Graphical Interface Generations", IFIP'89 World Congress, San Fransisco, 1989 .

[DRZ85] A. Demers, A. Rogers, F. K. Zadeck, "Attribute propagation by message passing", ACM Sigplan Symp. on Language issues in programming environments, Seattle, 43-58, 1985