A formal specification of appearance and behaviour of visual environments

by Fabio Paternó

The problem of a formal specification of visual languages has been addressed in several works. Most of these approaches only describe the correct placement of graphic objects on a screen. Icons are not, however, static elements. They behave dynamically in order to interact with the user, other icons and application processes. In order to make a complete specification of icons, it is important to have a formal approach which can describe the graphical status and the reactive behaviour depending on the generated events. A combination of process and data algebra for this purpose is proposed. This approach was applied to describe a visual environment developed by a set of common interaction techniques.

1 Introduction

There is currently much interest in visual languages [1, 2], as they are powerful tools for access by non-experts to the functionalities of an application. A visual program is defined by composing graphical symbols. Visual languages introduce interesting problems from a formal specification point of view. Whereas textual languages are defined by composing elements defined in a linear sequence, a graphical syntax is defined by composing graphical symbols that can have different relationships in the bidimensional space of the screen.

There are two main approaches to specifying and manipulating visual languages:

- by using structure editors, such as that in Reference 3. They are usually generated by automatic tools which receive the description of the visual language as input. The editor allows the user to select from a set of predefined choices in order to avoid mistakes;
- by using parsers, such as those in References 4 and 5. The user is initially free to specify graphical objects on the screens with a direct manipulation editor, and then a parser receives the developed graphical representations and recognises their relationship in order to obtain a correct specification. The parser usually works by using a specification of the visual language syntax developed by grammars.

Both these approaches propose formal specifications (developed by notation with a solid underlying mathematical support) to describe only the static appearance of the visual environment. This is too limited because it does not describe dynamic behaviour. The latter is very important because users need to be able to manipulate visual environments by direct manipulation, and this means working in a dynamic environment where it is possible to select the objects of interest and to have continuous feedback of the status of interactions.

Another reason is that graphical representations are more compact and more meaningful than textual ones. However, when many details have to be represented, graphical representations can also become difficult to interpret. Therefore, visual environments are often multi-layered, and are performed by using pop-up menus and similar techniques. A subset of graphical representations can be visualised, but only when the user generates a specific dynamic event is further graphical information provided. Conceptually, we have different layers of graphical representations, with user events allowing the selection of the part to be visualised. The first examples of these multi-layered approaches were menu trees.

More generally speaking, an icon, a graphical representation on the screen, is a dynamic object with a status (its graphical representation, application-dependent information, and so on) but which can also react to stimuli generated by the user or by other objects. Typical examples are scroll-bar, pop-up, pull-down menus, and buttons, dialogue tools to enter and modify texts etc.

This two-fold feature of icons has not yet been formally investigated. In this paper, we want to demonstrate that a combination of process and data algebra allows us to fully describe these types of objects. This makes it possible to specify visual environments completely, to improve our understanding of their behaviour and to define their properties.

There have already been proposals [6, 7] to design, in a structured and general way, graphical interaction, but these approaches may be too complex for designing...
iconic languages which only need a small set of interaction techniques and clear rules to compose them for a flexible, simple-to-use visual environment.

2 Related work

The problem of the specification of a visual language has been mainly focused on the definition of graphical syntax. This is defined by composing graphical items in the bidimensional space of the screen, which entails having operators for the composition.

Previous work at CNUCE [9] defined the graphical syntax for LOTOS (Language of Temporal Ordering Specification) [9] by exploring different solutions. Two methods for specifying GLOTOS based on the Pictor language were also described. They remind us that Pictor is not intended to be a general-purpose picture description language, and its operators do not claim to satisfy orthogonality requirements. The first technique is to extend the context-free grammar for textual LOTOS with additional productions and symbols which correspond to the Pictor operators. The extended grammar defines a set of Pictor terms which constitute the graphical syntax of GLOTOS. In the second approach, an abstract syntax for LOTOS is defined. The syntax of GLOTOS is given as a mapping between terms of abstract LOTOS and terms of Pictor. The mapping is defined with the predicate pi&, which associates an abstract LOTOS terms with its pictorial representation (expressed as a Pictor term). GLOTOS is defined as the interpretation of all those Pictor terms which are representations of abstract terms. In this paper, we face the opposite problem; instead of providing a visual representation of LOTOS constructs, we use LOTOS to describe a visual environment.

Golin [4] developed a graphical environment by using picture layout grammars that provide a formal definition of the visual language syntax. These grammars associate a production rule with a predicate and a semantic function. When the predicate, which is a condition between the graphical attributes of the items in the two sides of the rule, is verified, the semantic function is applied. This means computing some graphical attribute values from the left side by applying the function on the current values from the right side. In another work, Golin [10] focused on the problem of a visual environment in order to specify mapping from user actions to editor operations. He defined a visual environment for designing the dynamic behaviour of a visual environment, but he did not deal with the problem of a formal specification of the combination of appearance and dynamics of visual items.

Wing and Zaremski [11] use an application of the specification language Larch to describe a visual language, whose basic elements are boxes and arrows, for security configuration of file systems and general security policy constraints. Properties such as 'well formedness' of the picture (for example, all arrows must be attached to boxes) are also discussed.

Narayana and Dharap [12] proposed a formal specification of a Look manager. The model is based on the notion of texturing objects. The specification assumes an invariant relation between the logical display of objects and their layout on the physical screen. The look of the screen is characterized as an invariant ideal show relation. The specification is performed by the state-based specification methodology of Z. We feel that a formal notation with explicit concurrent constructs, such as LOTOS, is more suitable for describing a set of autonomous interaction techniques than notions such as Larch and Z.

Cinque et al. [13] addressed the problem of defining a dynamic visual environment. It is described by a set of states, each of which is associated with a set of icons representing application objects, a set of icons representing the actions that can be activated, a set of icons representing the warnings, and a finite set of presentation elements. By activating specific elements, it is possible to realize a state transition. This work presents an abstract approach, whereas what we want to realize is a formal specification of a more precise description of the dynamic behaviour of the interactive components of the visual environment.

Chi [14] presented a comparison and evaluation of four axiomatic approaches to the formal specification of user interfaces. He noticed that, of the notations he examined, only event algebra is capable of addressing an important issue such as concurrency in user interfaces; but this technique appears to be too unwieldy.

In this paper, we discuss the application of automatic tools to LOTOS specifications of user interfaces. There are other automatic tools which allow designers to reason on formal specifications; for example, an application of mural, a theorem-proving assistant and specification support tool, is presented by Fields and Elvang-Goransson [15]. However, we present a formal notation with data algebra, including the ACT ONE notation.

LOTOS was developed in the ISO environment to specify network protocols. It combines process algebra, deriving concepts from Hoare's CSP and Milner's CCS, with data algebra, including the ACT ONE notation.

The basic idea of LOTOS is to describe a system by its externally observable behaviour. A process can perform internal, unobservable actions and interact with other processes by means of external, observable actions (interactions). Actions are atomic entities that occur at interaction points or gates instantaneously, without consuming time. Processes can interact with each other by performing the common actions defined at their gates. A process definition specifies the behaviour of a process by defining the sequences of observable actions that can occur at the process gates. This behaviour is represented as a tree of actions the process can perform. It has synchronous communication. Different instances of the same process may differ by having different current gates. LOTOS is built from two components.

Basic LOTOS is a subset of the language and employs a finite alphabet of observable actions identified only by the name of the gate where they can occur.
cesses interact with each other by pure gate synchronisation, without value exchanges, so that there is no sense of direction in communication.

Full LOTOS extends basic LOTOS by structuring observable actions and process interactions in a more detailed manner. The major advantage is the enhancement of synchronisation with value passing, thus providing for interprocess communication. Actions are identified by a gate name and by a list of zero or more values offered at that gate. Data values are expressed by including type definition in the specification. LOTOS type specification is derived from the specification language for abstract data types ACT ONE. Data types are consequently specified by an algebraic notation consisting of a signature and a list of equations. The signature is the set of sorts and of total operators over the sorts of a data type. The equations are the set of constructs which state that two syntactically different terms denote the same value.

The main operators in a process behaviour definition are

- action prefix; for example, a;B means that the process can only perform a and then behave like B.
- choice; for example, B1|B2 means that the process can act as B1 or as B2.
- hiding; hide g1, ..., gn in B is a process which can perform any of B's actions which do not make use of gates in (g1, ..., gn), any action occurring at one of these gates is hidden and transformed into an i-action; i denotes the internal, unobservable action.
- guarding; such as [e] → B, if the Boolean expression e is true, then B is performed.

Processes can be composed in different ways:

- P1| [g1, ..., gn]| P2: it means that P1 and P2 can synchronise on gates g1, ..., gn, where there may also be value passing.
- P1 || P2: it means pure interleaving; the two processes never synchronise.

Fig. 1 The stages of user-interface development

Enabling and disabling among processes can also be performed.

Data and process algebra mainly interact in the definition of processes, where the data types of the parameters which define the status of the process, and the messages that can be received in the communication gates, have to be indicated by the data algebra.

This formal notation was conceived to specify network protocols, but it can also specify each type of concurrent system. Interactive graphics systems are intrinsically concurrent systems consisting of different interaction techniques that are autonomous processing entities. The trend is to increase their parallelism in order to better support the possibility of interacting with multiple users who use multiple devices and with application multi-tasking. This implies that sequential languages, such as Pascal, cannot describe them satisfactorily.

LOTOS is particularly suitable for specifying these types of systems because its two-fold aspects also match the two-fold aspects of the visual environment; process algebra can be used to describe the dynamic behaviour of user interfaces, and data algebra can describe their state, which determines their appearance and also how they are modified depending on the interactions performed.

4 The graphical user-interfaces development environment

In order to appreciate the support of the application of formal methods, we need to understand how they can be located in the stages of the user-interface development. A set of automatic tools should therefore be available to manage the performed specifications. Indeed, a mere paper-and-pencil approach cannot provide meaningful results, especially if the systems considered are large. We apply LITE [17] to visual environments specifications. In this way, better results can be obtained because LITE pro-
vides a large set of tools which supports editing, static semantics checking, report generation, simulation, compilation, transformation, verification and testing.

The tools that we found the most useful for a user interface development environment (Fig. 1) are

- the verification of correctness from a syntactical and static semantic point of view.
- the simulation of the dynamic behaviour of the specification and the interactive visualisation of its event tree.
- the verification of its properties by a temporal logic checker, which checks whether properties defined in action-based temporal logic are verified on the automation generated by the LOTOS specification. These properties are also useful because they summarise aspects of the user interface which are not obvious from the specification itself.

In this way, we gain a deeper insight into whether user interaction performs the desired semantics by processing the LOTOS user-interface specifications. Then, if the specification corresponds to the designer tasks, a refinement into a programming language can be performed in order to obtain the system with which the user will interact.

In this work, we concentrate on the development of the formal specification of the visual environment and provide some examples of application of the available tools in order to illustrate this methodology. We also outline some possible results.

The only problem found in the formal approach is that formal notations are not very easy or clear to read and understand. In order to overcome these problems, we have tested some graphical tools [8]. We have noticed that a graphical syntax of a formal notation, such as Graphical LOTOS, where we mainly have a graphical symbol associated with each formal notation operator, often provides graphical representations that are complex and difficult to interpret. This happens especially if the specification considered is large, and so sometimes the textual specification is clearer. A more interesting approach would be to associate graphical symbols with sequences of formal notation constructs which are usually applied to specify user interfaces.

5 A case study: a simple data-flow language

Below we discuss a general approach, which is application-independent, describing a visual environment composed of interactive icons. We consider NetGIS [18] as a case study. It is a previously defined visual environment for GIS applications, and here we discuss the results obtained by performing its LOTOS formal specification. Two approaches to a formal specification of this language were explored [18]: common BNF grammars and picture layout grammars (PLG). Both were unable to describe the dynamic behaviour of the environment, how the user interface reacts to the events generated by the user, such as button pressing or mouse moving, because they were only used to describe the correct compositions of graphical symbols on the screen. In fact, the BNF grammar whose terminal symbols are associated with the available graphical symbols allows us to indicate which networks of icons are correct with respect to the visual language definition.

![Fig. 2 The interactive subfields of an Icon](image)

This grammar was used in the implementation of the structure editor that allows and guides the user to compose the desired visual program. The PLC have a different approach; they need a set of previously defined graphical symbols, and then the user can create instances of graphics symbols and place them in an unconstrained way on the graphic area. The resulting graphic representation will be used in order to verify whether it corresponds to the logical specification. Neither approach considers the specification of the interactive aspects of the visual environment; these are the possible events generated by the user actions and how the environment reacts to them. In Section 6, we describe three simple examples.

The basic idea of NetGIS was that, in order to make the translation from the user tasks to the system functionalities easier, it is important to associate an icon with a typical medium level of abstract GIS function. The icon provides a more immediate representation of the related functionality and hides different details which are typical of textual syntax. This allows users to work with graphical representations that are closer to their conceptual model of the problem and its possible solutions.

In this case, a visual program, which is a composition of icons representing the corresponding composition of GIS functionalities, is obtained by following the data-flow approach; the icons are composed of links that represent the data exchanged among the application functions corresponding to them. A network is a correct composition of graphical elements that can be executed. In order for a connection between two icons to be correct, it has to verify two conditions; the data types associated with the input and the output ports have to be compatible, and the input port must not already be associated with another link.

The editor allows the user to create new instances of icons and links, move them, modify the status information of the selected icon, to delete an icon, to save and restore a visual program, and other common functionalities.

In this environment, an icon is more than a passive
matrix of pixels; it has an associated set of interactive fields (Fig. 2): mover, to move the icon; info, to display information on the related functionality; more, to visualise and modify parameters of the related application functionality (this can be very useful if the user wants to configure a GIS function for specific purposes); and input and output ports, one for each communication port of the related functionality.

In this case, icons are graphic objects with a fixed size, whereas links can modify their size depending on the interactions performed by the user.

A first set of eight basic modules was defined [18]. The main difference between them from a user-interaction point of view is given by the available communication ports. They can be classified into four groups, depending on the I/O ports:

- 0 input port and 1 output port (Set, to select a map).
- 1 input port and 0 output port (Display, to visualise a map, Report, to provide information on a selected element).
- 1 input port and 1 output port (Edit, to edit a map, Topology, to generate correct topology of a set of data, Buffer to provide buffer area).
- 2 input ports and 1 output port (Overlay, topological overlapping among two maps. Near, to compute closest points).

6 Examples of dynamic situations

One of the strengths of LOTOS is the possibility of specifying the dynamic behaviour of concurrent systems. This is one of the reasons why it is very useful for specifying visual environments where, for example, often at a given time a user can select a non-deterministic choice from a set of possible actions. In this Section, a few examples of possible dynamic situations in the considered visual environment are illustrated.

Currently, the most common way to interact with graphical user interfaces is with a mouse. Events such as pressing a button are directed by the underlying window system to the interaction techniques associated with the related window. This means that in the following specification each process has its own specific button press event, which is related to the pressing of the button when the cursor is in its graphic area.

The visual environment is defined by a set of processes. We define one process related to the editor and one process for each of icon. At each time, the visual environment is described by the composition of the editor process and the processes associated with the icons that have been chosen.

Dynamic aspects of the visual environment to be described include:

- creation of a new icon (Fig. 3). This is performed by selecting an icon from the set of available icons in the left-hand column, and then by indicating a position in the work area where the new instance is to be placed.
- activation of a pop-up menu. In Fig. 4, an example is shown where the information field of the icon is selected, followed by the possibility of inserting or deleting text by events generated by keyboard pressing, or deactivating the pop-up menu by selecting the quit area.
7 The visual editor

The editor process has a state that consists of the initial position and the list of the example icons. It manages the part of the main window related to the set of available icons and the work area where the visual program is composed. The set of specific commands is managed by a process which is activated when an icon is selected. When it receives a button press event, it behaves differently depending on the area selected:

- if it is quit, it deactivates itself.
- if it is an icon associated with a GIS functionality, it waits for a new button press event in the work area where the new icon is located. A parallel composition between the visual environment and the new icon is then activated.
- if it is a link, it waits for two button press events, one of which has to be on the output port of an example icon and the other on the input port of another icon in order to activate the parallel composition with the new link process.

The LOTOS constructs below are written in italics.

```lotos
process editor [bpress, ed, popin, popst, ll, li, lo, kpress, kpressin, kpressst, ap, bpresspop, bpressl, cmd] (Pos : Icon, c : List_Icons, n : Nat) : noexit :=
  bpress?P : Point;

endproc
```

Here is a similar behaviour to the previous choice for each type of icon related to a GIS high-level functionality.

```lotos
[is_in_work_area(P, Pos) → editor[bpress, ed, popin, popst, ll, li, lo, kpress, kpressin, kpressst, ap, bpresspop, bpressl, cmd] (Pos, c, n)]

[is_in_link_area(P, Pos) → bpress?Pd : Point; cmd? bl1 : Bool?ln : Link;
  [ln eq lin] → [is_in_work_area(P, Pos) → editor[bpress, ed, popin, popst, ll, li, lo, kpress, kpressin, kpressst, ap, bpresspop, bpressl, cmd] (Pos, c, n)]]
```

Software Engineering Journal  May 1993
When a link is selected in the editor, the user selects two link fields in two icons; the first has to be an output port and the second an input port. If the input port is already associated with a link (bi2 is false), the request fails because a GIS function at a given time can only receive data from one other function. If the interaction fails, the user has to select the link field of the editor again if they still want to create a new link instance.

8 An interactive icon

In our case, an icon is a rectangular collection of pixels, each indicating a colour by selecting an item in the colour map. Once an icon has been defined, its appearance cannot be modified; only its position on the screen can be changed. The process icon has a state which consists of the description of the current position of the icon (defined by a point associated with the upper left corner), information to explain to the user the related functionality and parameters which depend on its semantics. It can communicate with the user for three purposes: to move from one point to another in the work area; to enable a pop-up menu to show further information related to the functionality associated with the icon; and to select the icon in order to delete it or execute the related functionality.

A pop-up menu is used in two cases; to visualise information on the related functionality or on its state (which is defined by its class, the semantic task associated, a set of parameters to modify or customise the current associated semantic function, and a Boolean to indicate if the associated connections have been defined).

Next we consider, as an example, the process associated with the overlay functionality. An overlay process communicates with seven objects: two pop-up menus, three links, the application process and the editor process. Its header indicates the communication gates: the most important are bbox (for receiving button press events), ed (for receiving commands related to the icon), ap (for receiving the results from the related application functionality), l1 and l2 (for receiving new input data for the related application functionality), lo (for transmitting the results of the related application functionality), and cmd for communicating with the editor process.

The parameters which define the state are of Icon type, indicating position and type of the icon; inf_list type, indicating information to remind the user which are the associated functionalities and to show the state of the application functionality parameters; Boolean type, indicating if the related links have been instantiated; and App_data type, indicating the current input values for the application functionality.


This process can initially receive a button press event. Its behaviour then depends on which subfield is related to the position received with the button press event. If the move area has been selected, then it waits for a new button press event and indicates the new position that is inserted in the process state. This is communicated to the connected links (there is a Boolean for each of them indicating if they are instantiated) with the icon, and it then calls itself recursively with the parameter associated with its current updated position.
Fig. 6 An example of simulation of dynamic behaviour

If the button press is related to the information or the status area, then a parallel composition is activated with the processes related to the popup menu. These processes are activated with a state defined by information derived from the current state of the calling process and a position that depends on the position of the calling process. The popup menu is placed in a fixed position, depending on the position of the calling icon. If a field related to a link is selected, this means that a new link has been instanced by connecting the considered icon, and a Boolean is set to true in order to keep track of the modifications. In the case of input links, if that link is already allocated for another connection, a popup menu with an error message is activated. In this case, we can use the interleaving operator (||) because icon and popup menus must not synchronise themselves.
Above we have illustrated examples of the dynamic behav-
ior of the components of a visual environment. To de-
tine them, we used some algebraic data types in order to
define the parameters composing the processes' state and
the data types that they can receive.

Below are a few examples which adopt the algebraic
data type part of LOTOS to describe some data types
used in the specification.

TYPE POINT is REAL

sorts Point

opns mk_point: Real, Real → Point
get_x : Point → Real
get_y : Point → Real

endtype

In our processes, an icon data type is only defined by
one point and one icon type. The entire icon object is
defined by a process such as that described above. We
have a set of predefined icon types, for each of which there
is an associated matrix of pixel values defining its appear-
ance.

TYPE ICON is POINT, ICON_TYPE, INTEGER,
NATURAL_NUMBER, BOOLEAN

sorts icon

opns

mk_icon : Point, ICON_TYPE → Icon
pos_icon : Icon → Point

is_in_move_area : Point, Icon → Bool

is_in_info_area : Point, Icon → Bool

is_in_linkdo_area : Point, Icon → Bool

is_in_linkdi_area : Point, Icon → Bool

is_in_more_area : Point, Icon → Bool

is_in_select_area : Point, Icon → Bool

expr

forall P, P1 : Point, ic : ICON_TYPE

bool

The operations to select a subfield in the icon are defined
in the same way; they compare the position received with
the predefined area related to the specific subfield which is
dependent on the position of the icon. For example, the
move field is defined by the rectangle with its lower left
corner indicated by the point (get_x(P) + move_minx,
get_y(P) + move_miny) and its upper right corner indi-
cated by (get_x(P) + move_maxx, get_y(P) + move_maxy),
where P is the position of the upper left corner of the icon
and P1 is the point selected by the mouse.

is_in_move_area(P1, mk_icon(P, i)) = (get_x(P1) ge
(get_x(P) + move_minx)) and (get_y(P1) ge
(get_y(P) + move_miny)) and (get_x(P) le
(get_x(P) + move_maxx)) and (get_y(P) le
(get_y(P) + move_maxy))

... the other functions are defined in a similar way.

endtype
Now we can see some other processes of our visual environment. A pop-up menu is an icon which is dynamically activated and deactivated. It can choose between two behaviours, depending on the events received; receiving new information from the user in the form of character sequences (kpress) or a button press (bpress) in the quit area that deactivates the icon after transmitting (pop gate) the update state to the application.

process pop_icon[kpress, pop, bpress] (P1 : Icon, st : Inf_list) noexit:=
  kpress?inf: Inf_list; (Inf eq del) → pop_icon[kpress, pop, bpress](P1, delete(st))
  [[Inf ne del] → pop_icon[kpress, pop, bpress](P1, update(inf, st))]
  [[bpress?Point; (P, Pout)] → poplst; stop
  [[P = _in_quit_area(P, P1)] eq false] → pop_icon [kpress, pop, bpress](P1, st))
endproc

A link is represented by a process with a status defined by the two end-points. It can receive events indicating that the user wants to modify its geometry; that the sender functionality wants to transmit its results to the next one; the two end-points. It can receive events indicating that the user wants to modify its geometry; that the sender process pop-icon [kpress, pop, bpress] (P1, noexit:=
  kpress?inf: Inf_list; (Inf eq del) → pop_icon[kpress, pop, bpress](P1, delete(st))
  [[Inf ne del] → pop_icon[kpress, pop, bpress](P1, update(inf, st))]
  [[bpress?Point; (P, Pout)] → poplst; stop
  [[P = _in_quit_area(P, P1)] eq false] → pop_icon [kpress, pop, bpress](P1, st))
endproc

9 The simulation of the dynamic behaviour of the visual environment

The dynamic behaviour of a user interface cannot be completely examined by the user because the external appearance only provides partial information. The user interface is realised by the composition of different processes that can synchronise themselves in different ways before providing graphical representations for the user. By applying specific automatic tools, we can monitor and study the internal dynamic behaviour without implementing and executing the visual environment.

Fig. 6 shows an example of processing that can be performed on the specification with these automatic tools. We consider a basic LOTOS version of the specification in order to concentrate on its dynamic behaviour. This is described by the event tree associated with the specification. The event tree provides a complete description of all the possible evolutions of the specified system. If we select an event in the related tree, all the events between it and the root, which define the history of interactions in order to arrive at the given interaction, can be considered as a precondition, and the set of the next events are the possible postconditions from an interactive point of view (the available interactions at that time). In the case represented in Fig. 6, the designer selected the overlay process specification and only asked for its event tree for a one level depth. A column with the first set of possible events appeared (bpress, ed, ap, l1, l2), and then the user selected one event (ed) and again asked for a list of possible events with depth one. In the same way, they then selected four events (before three times ap and then lo) from the possible choices which become available at each level.

The dynamically selected branch of the events tree represents one possible evolution within the time of the specified system. In this case, it describes what happens when the user selects the run command for the selected overlay Icon; its related process sends the current data to the application process, which then returns the result which is transmitted to the output link. The window path in Fig. 6 indicates the sequence of selected events.

This type of processing is very useful for evaluating the dynamic behaviour of the specification. It is possible to simulate the events generated by the user and the application, and to study how the system reacts to these events. Consequently, for example, we have a clear indication of the actions that the user has to perform in order to fulfill a specific task.

10 Verification of the formal specification

The LITE environment can also demonstrate properties expressed in action-based temporal logic (ACTL) on the automaton corresponding to the LOTOS specification. This tool is very useful because it allows us to reason on the specification and to summarise synthetically the features of its dynamic behaviour. It is beyond the scope of this paper to explain ACTL. Interested readers should see the work by de Nicola et al. [19].

Software Engineering Journal May 1993
ACTL is a branching time temporal logic (so it allows us to discuss the properties of all the possible observations of a system), which permits the expression of properties of systems in terms of the actions that they can perform instead of their states.

In order to show how it is possible to automatically verify property expressed in ACTL, we can consider the specification of the pop-up menu in our visual environment. The corresponding automaton is shown in Fig. 7.

We applied the automatic tool in order to check two properties on the pop-up menu specification as examples of the possible results that can be obtained by it. One property asserts that, for all the possible futures (operator A), it is always true (operator G) that if the bpress event is performed, then there exists a future where the pop event will be performed:

\( AG \{ \text{bpress} \} \text{E} \{ \text{true} \} \cup \{ \text{pop} \} \text{true} \).

The other property says that, for all the possible futures, if the kpresso event is performed, then there exists at least one future (operator E) where the pop event will not occur:

\( AG \{ \text{bpress} \} \text{E} \{ \text{true} \} \cup \{ \text{pop} \} \text{false} \).

11 Conclusions

We have illustrated a method of formally describing the interaction among users, the application and interactive icons. The visual environment can thus be described completely and precisely, and in its dynamic and concurrent aspects. This allows us to reason about the specification. It is thus possible, for example, to verify whether there are equivalences among user interfaces with respect to user-generated events, or whether there is a deadlock situation or other properties. This topic is studied in depth elsewhere [20, 21].

Approaches such as picture layout grammars can only indicate if a graphical representation is correct with respect to the language definition. In our case, this means that an example set of icons and links was highlighted and composed correctly on the graphics screen. The layout description can also be obtained by our specification because, for example, an icon process does not allow the user to associate two links with an icon input port. In addition, our approach can formally define how users can interact with the visual environment and how their actions can affect the appearance of the visual environment. This is very important as, in addition to verifying the correctness of the dynamic behaviour, the visual environment can also be completely specified in order to provide the user with a complete documentation.

The two-fold aspects of LOTOS (process and data algebra) can be used to describe the two-fold aspects of visual environments (appearance and behaviour). Different styles of specifications are possible, depending on how the two components of the formal notation are used. Future work will be devoted to identifying the most suitable style of specification for this application area.

12 References


The paper was first received 1 April and in revised form 6 November 1992.

The author is with CNU/CNR, Via S. Maria 36, 56100 Pisa, Italy.