On the Use of LOTOS to Describe Graphical Interaction

F Paterno' & G Faconti

CNUCE-Institute of CNR, Via S.Maria 36, 56100 Pisa, Italy.

Tel: *+39 50 593289*

Fax: *+39 50 589354*

EMail: *paterno@icnucevm.cnuce.cnr.it*

This paper discusses a formal specification of a model of a graphical interaction object by using the LOTOS notation and the possible results that we can obtain from this approach. With this model the relationship between input and output functionality can be addressed. A User Interface System, which manages dialogues between the user and the application, may be described as a composition of instances of graphical interacting objects. Examples of common graphical interactions are described following the proposed abstract model for graphical interaction objects and by using the LOTOS notation. Application of automatic tools to the performed specifications is discussed.

Keywords: User Interface System, Formal Methods, Interactive Systems, Graphic Input Models.

1. Introduction

In the last few years there has been a growing interest to provide user interfaces with a better usability. This entails them supporting multiple dialogues, which can be active at the same time. In addition, the relationship between input and output functionality has to be clearly identified since they are strictly connected. This means that the design of these types of systems will become very complex as they must be both structured and flexible. It can be achieved by using a formal notation to demonstrate the correctness of the specification and its properties. So if we want usable user interfaces, we need to perform their formal specification in order to understand and to manage their underlying design complexity. In this paper we present the refinement of a description of a model of a graphical basic interaction, along with its formal description using the LOTOS (Bolognesi & Brinskma, 1987) notation.

The problem of designing graphical user interfaces can be described on different abstraction levels:

- The *conceptual*: Where the main functionalities that should be present are identified.
- The *architectural*: Where the basic components and their relationship are identified in order to carry out these functionalities.

The first abstraction level has been tackled at different workshops (Seeheim (Pfaff, 1985), Seattle (Olsen, 1987), Lisboa (Duce et al., 1991), Arch (Bass et al., 1991)). At the first workshop a classical subdivision of User Interface Systems (UIS) into three layers (Presentation, Dialogue, Application) was proposed (with concepts from the linguistic approach). This approach was found correct but too generic. For this reason in (Duce et al., 1991) they were split up into a set of three types of objects: *interaction objects*, which allow the user to interact on a specific media; *transformation objects*, so that the interaction objects can be controlled and constraints on their behaviour can be defined and *monitor objects* which provide services such as history of transactions. In the latter (Bass et al., 1991) the Seeheim Model was revised and a further subdivision into five layers was proposed: the interaction toolkit component; the presentation component; the dialogue component; the domain adaptor component and the domain specific component. With this approach several toolkits and a domain specific component can be integrated in one instance of UIS.

There have also been several proposals for the second abstraction level: the first was Smalltalk (Goldberg & Robson, 1983), which proposed the model-view-controller paradigm. Other interesting proposals were developed by Myers (Myers, 1990), who indicated a small number of interaction classes, each one encapsulating a specific interactive behaviour, which permitted the description of a broad spectrum of graphical interaction. Coutaz (1987) developed the PAC (Presentation, Abstraction, Control) model.

The problem of the formal notation of user interfaces and graphics systems has been dealt with following two types of approaches: by using notations developed for this specific purpose or by using general purpose formal notations. In the first class are such languages as Agent (Abowd, 1990), which combines CSP and Z constructs and Manifold (Soede et al., 1991) which has been used to describe the input model of graphics systems. Examples of the second approach include: CSP to describe hierarchical input devices (Duce, ten Hagen & van Liere, 1990); Temporal Logic to investigate properties of graphic interaction (Faconti & Paterno', 1992); Z for a configurable approach to graphics systems (Arnold, Duce & Reynolds, 1987); VDM for some aspects of GKS (Duce, Marshall & Fielding, 1988); OBJ for the revision of GKS (Duce & Damnjanovic, 1992); and four axiomatic approaches (Chi, 1985) are compared and evaluated by using a specific example of user interface as a test case.

We believe that LOTOS is well suited to specify UIS. It has concurrent constructs. This is important to better specify concurrent systems such as UIS, which has concurrent behaviour for different reasons:

- They can simultaneously receive requests from the application and from the user.
- Both of them can simultaneously generate requests at different points in the related interfaces because the application can be multitasking and the user can activate several logical input devices at the same time.

- The data received by the UIS can be processed in parallel and all data can receive different levels of processing through different interaction objects.

LOTOS includes data algebra, in addition to process algebra, so we can also describe the state of each graphical interacting object and the processing that it applies according to the realized interaction with the external environment.

Our approach should make it possible to integrate the description of common interaction techniques with the manipulation of structured graphical objects, which can receive processing at different abstraction levels. In order to provide a general framework for the available graphics elaborations we consider the Computer Graphics Reference Model (ISO/IEC DIS 11 072, 1991). It defines five environments:

- *Construction*: Where the application data to be displayed is prepared as a model from which specific graphic scene can be produced.
- *Virtual*: Where the geometry of the output primitive are completely defined.
- *Viewing*: Where a specific view of the scene is taken and the result of the projection is transmitted to the lower level.
- *Logical*: Where the output primitives are associated with a complete set of properties in a device-independent way.
- *Physical*: Where the image is presented as a display for output to a specific device.

There is an architecture common to all the environments and in each of them, the input and the output pipeline can interact. In order to integrate user interaction with sophisticated graphics processing we aim to design a UIS like a graph of interactors with each one logically placed at one of the five abstraction levels depending on the type of graphics processing that they carry out. The basic idea is to extend a hierarchical composition of input devices enabling each of them to have output functionality.

Section 2 briefly introduces the LOTOS notation. Section 3 describes our refinement of an interactor architecture. In Section 4 we provide a discussion on the role of formal specification in the development of user interfaces. Section 5 details how to specify an interactor, firstly by using basic LOTOS and showing the results obtained by applying automatic tools to it and then by using Full LOTOS. Section 6 describes some common graphical interactions in order to exemplify possible results within the developed approach. Finally, some concluding remarks are reported.

2. The LOTOS Formal Notation.

LOTOS is a specification language which was developed within the International Organization for Standardization (ISO) environment in order to specify network protocols. Its features enable it suitable to describe concurrent systems like UIS. It combines the process algebra inherited from CSP and CCS with the data algebra provided by ACT ONE. Basic LOTOS is considered to be only the concurrent part of the notation.

The basic idea of LOTOS is that systems can be specified by defining the temporal relationships among the interactions that constitute the externally observable behaviour of a system. A system is seen as a set of processes. Each process may interact with its environment via interaction points called gates, i.e. it performs observable actions at the gates and can also perform unobservable (internal, hidden) actions. An observable action consists of

offering or accepting zero or more values at a certain gate. An interaction may occur when two or more processes are ready to perform the same observable action(synchronous communication). It may involve data exchange and is an instantaneous event.

Process behaviours are described with algebraic expressions, called behaviour expressions. Complex behaviours are expressed by composing more simple behaviour expressions via the LOTOS operators, such as sequentiality, parallel composition and disabling. In a process definition the specifier must indicate the process identifier, a formal parameter list and a behaviour expression (and the definition of the data types and the processes that it uses).

The main operators in a process behaviour definition are:

- *Action prefix*: For example `a;B` which means that the process only performs `a` and then behaves like `B`.
- *Choice*: For example `B1[]B2` which means that the process can act as `B1` or as `B2`.
- *Hiding*: `hide g1, ..., gn in B` is a process which can perform any action of `B` that does not utilise gates in `(g1, ..., gn)`, any action occurring at one of these gates is hidden and transformed into an i-action.
- *Guarding*: Such as `[e] -> B`, is carried out by evaluating the Boolean expression `e`; if it is verified `B` is executed.

When a composition between two processes has to be defined there are three possibilities:

- *Interleaving*: Denoted by `A|||B`, means the processes are independent of one other and any interleaving of their actions is possible, which means the processes never synchronize.
- `A|[f,g,h]|B` explicitly indicates the gate where the independent processes have to synchronize.
- *Full synchronization*: `A||B`, here the two composed processes are forced to proceed together.

It is also possible to carry out:

- *Sequential composition*(Enabling): `B1 >> B2`, where the behaviour of process `B2` is enabled if and when the behaviour of the other process reaches a successful termination.
- *Disabling*: `B1 [> B2`, where the behaviour of process `B1` can be disabled at any time (except after successful termination) by performing an action of the other process.

The process communication can be carried out in different ways, the more important ones are:

- *Value passing*: If we have process `A` realizing `a!3;` and process `B`, `a?x:int;`, the result of their synchronization is that the value 3 is passed to `B` in the variable `x`.
- *Signal matching*: Where for example, process `A` realizes `a!x1`; and process `B`, `a!x2;`, the two processes sharing the event gate a may attach signal values to events and the corresponding events take place if these values coincide `(x1=x2)`.

The synchronization mechanism in LOTOS has some distinctive features: there may be more than two processes involved, and they are not fixed, but can vary dynamically during

the evolution of the system, so that no participant knows exactly which are its partners at any point in time. A process which is ready to participate in one or more synchronizations does not necessarily wait until one of them takes place, but may execute local actions that render it no longer ready for some of the synchronizations for which it was once ready.

The union between process and data algebra is mainly performed in two ways:

- By defining the state of a process being described (a set of related parameters whose current value must be indicated each time the process is instanced).
- By defining the data types received in the communication gate which receives messages.

With the available process and data algebra, different styles of specification are possible, depending on which component has more weight. The choice of the style is determined by different considerations, foremost are the aspects which the designer wishes to stress.

3. An Interactor

In (Faconti & Paterno', 1990) there is a first description of an interactor, using ECSP, an extension of Hoare's CSP. It describes a basic graphical interaction as a composition of five functionalities:

- *The collection*: Where a high level description of the external appearance is defined.
- *The feedback*: Which produces the external appearance.
- *The measure*: Which builds a higher level input data.
- *The input trigger*: Which indicates the moment when the measure must apply its function and to provide its result (a higher level input data) to the control.
- *The control*: Which delivers the produced input data to other interactors or to application processes.

By working on this model we point out the need for some modifications. An output trigger condition must also be explicitly introduced to indicate the moment when the high level description of the output part must be interpreted and transmitted to the feedback and the measure. Another modification with respect to the previous specification was to consider the trigger as an event instead of a process. This is because the trigger is a useless process and introduces inconsistency: by analyzing the events tree of the previous specification we find different traces that allow the same trigger to validate different data. To remove the inconsistency and also obtain a simpler specification, the triggers are not considered as processes but as control events of the related process. The resulting architecture is described in Figure 1. In the next paragraph the interactions among the component processes are described and discussed in detail.

We can further model an interactor behaviour by means of specific control objects, the transformation objects. They control the four input channels of the related interactors and thus determine whether they will react or not to the receiving events. This paper does not address these (important) aspects.

Each interactor has two well-defined behaviours: the *internal* one and the *external* one. The former is the processing that the interactor performs to produce higher level input data

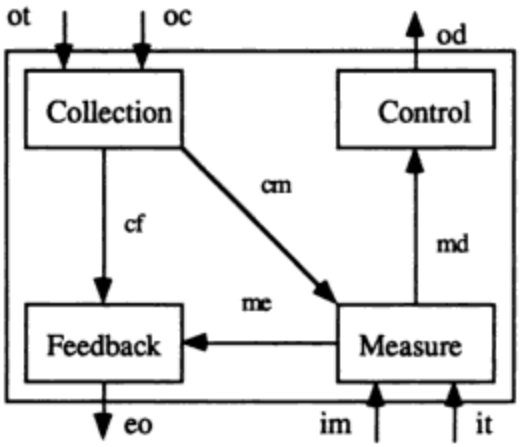

Figure 1: The Architecture of an Interactor

(in od gate). The latter is the processing used to generate lower level output data (in eo gate). The two behaviours interact inside an interactor in two ways:

- The Measure needs output primitives from the collection to build its input data.
- The Measure communicates to the feedback its status in order to provide an echo towards the user.

An interactor belongs to one of three classes: input-output, input or output, depending on whether it has full functionality or only the internal behaviour or only the external behaviour. An interactor which only has input can generate output but just for echoing to the user the state of the device but it has not the Collection process. An interactor which only has output is composed of just a Collection and a Feedback process.

A User Interface System can be obtained by the composition of interactors. There are two composition operators:

- An external composition operator which composes the external behaviour of two interactors that means to pass the result of the feedback process of an interactor as input of the collection of the other process.
- An internal composition behaviour, that is composing the internal behaviour of two interactors, by passing the result of a measure function of a process as input to the measure function of another interactor.

By applying these composition operators, we can obtain UIS with a very complex logical structure, with a lot of interlinks. We can also describe multi-level feedbacks with it.

4. The Role of Formal Specification in the Development of User Interfaces

Before presenting the LOTOS specification of a general description of a graphical interaction and some examples of its instances, we discuss the role of formal specification in the

development of user interfaces. This is important because it allows us to demonstrate the correctness of the specification and its properties while it provides a clear, precise, unambiguous specification of the user interface functionalities and behaviour.

In this paper we present the first results of our application of the LOTOSPHERE Integrated Tool Environment for LOTOS (LITE) (Caneve & Salvatori, 1991) on our LOTOS specification of interactors. This tool provides a rich set of functionality, for example:

- A syntax checker and a static semantic checker to verify specification correctness.
- A simulator allowing users to visualize the event tree associated with the specification (it is also possible to interactively indicate which branch of the event tree to unfold).
- A drawing of the automaton associated with the specification.
- Verification of the validity of temporal-logic formula on the performed specification.

Formal notations make specifications more precise but not more immediate due to their mathematical nature. In order to provide the user interface designer with a working environment closer to his conceptual representation of the user interfaces, it is important to create a visual environment that is defined by graphical representations which are associated with the constructs of a formal notation describing user interfaces. In (Paterno' & Faconti, 1992) there is an example of a graphical syntax of user interfaces designed on a ECSP specification of a first version of interactors.

The resulting environment for the development of user interfaces allows the designer to work by direct manipulation on graphical representations of interaction objects. The results of the specification are automatically translated into the formal notation. This allows the designer to simulate its dynamic behaviour, to verify the related properties without implementing the specification and to perform further studying. Then, if the behaviour of the specification corresponds to the goals of the designer, it may be refined in a programming language in order to obtain an executable system to provide to the end user.

5. A LOTOS Specification of an Interactor

We start from a description in basic LOTOS of an interactor. With it we can apply some of the available automatic tools. In the interactor specification we hide the events generated among its components in order to make observable only the events generated to communicate with the outside.

```
specification Interactor[ot, oc, od, eo, im, it] : noexit
behaviour
    hide me, cf, cm, md in
((COLLECTION[ot, oc, cf, cm] |[cf]| FEEDBACK[cf, me, eo])  |[cm, me]|
  MEASURE[im, it, cm, me, md]) |[md]| CONTROL[od, md]
where
```

The process collection has a higher level description of the external appearance of an interactor. This can be received by gate `oc`. When the gate receives a trigger event, `ot`, the information is transmitted to the measure, `cm`, and to the feedback, `cf`, and then returns to the initial state. If, after the activation of the process, the first event is that of the trigger, a default collection is transmitted to the other processes.

```
process COLLECTION[ot, oc, cf, cm] : noexit :=
    oc; COLLECTION[ot, oc, cf, cm]
    [] ot; cm; cf; COLLECTION[ot, oc, cf, cm]
endproc
```

The feedback provides an echo, **eo**, of the received input measure function value, **me**, or it receives data from the collection, **cf**, and visualize them, **eo**, only after having synchronized, **me**, with the measure, which in the meanwhile, has been updated on the new information that it too has received from the collection.

```
process FEEDBACK[cf, me, eo] : noexit :=
    cf; me; eo; FEEDBACK[cf, me, eo]
    [] me; eo; FEEDBACK[cf, me, eo]
endproc
```

The measure can receive:

- input data from the outside, **im**, and then provide information for echoing, **me**;
- the input trigger event, **it**, which is then followed by the communication of the current result of the measure function to the control, **md**;
- new information from the collection, **cm**, which is used to build the measure function, and then to communicate with the feedback, **me**, for echoing the new state.

```
process MEASURE[im, it, cm, me, md] : noexit :=
    cm; me; MEASURE[im, it, cm, me, md]
    [] im; me; MEASURE[im, it, cm, me, md]
    [] it; md; MEASURE[im, it, cm, me, md]
endproc
```

The Control process is only used to deliver, **od**, to the outside the higher level input data, produced by the measure, **md**.

```
process CONTROL[od, md] : noexit :=
    md; od; CONTROL[od, md]
endproc
endspec
```

This specification was shown to be correct by analyzing the syntax and the static semantics with automatic tools. One of the reasons for using formal specifications in the design of user interfaces is so that we can reason on the specification before implementing it. This is useful to demonstrate properties and to test whether the effective behaviour corresponds to the needs of the designer. We show first examples of how a deeper insight into the behaviour of a graphical interaction can be obtained with processing based on the LOTOS formal specification.

By applying tools (Caneve & Salvatori, 1991) we find that the minimized interactor specification has 50 states and 143 transitions. An automaton, corresponding to the given one, minimal with respect to strong bisimulation always has 46 states and 133 transitions; a corresponding automaton, minimal with respect to weak bisimulation, has 32 states and 102 transitions. Figure 2 shows the state transition diagrams of the three main interactor functionalities: feedback, which has three states and four events; measure, which has three states and five events; and collection, which has three states and four events.

The tool automatically generates the simulation events tree, derived from the specification. Figure 3 shows the first three levels of the resulting tree. The events represented in brackets

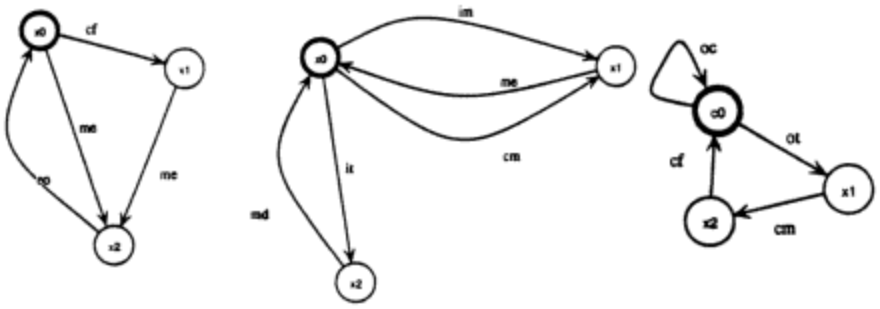

Figure 2: State Transition Diagrams of Feedback, Measure and Collection

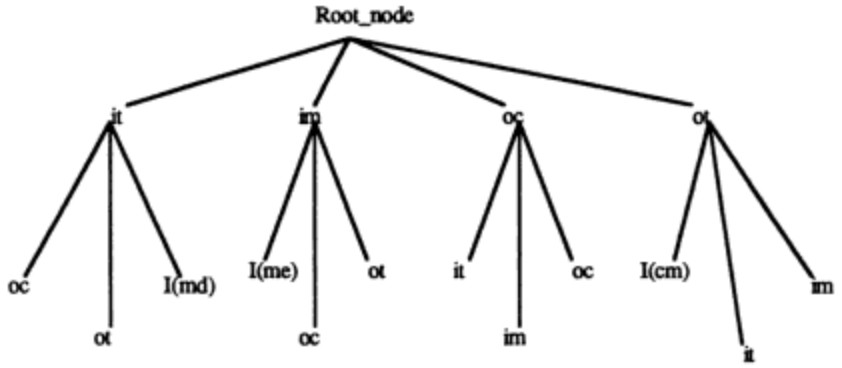

Figure 3: The First Levels of the Event Tree of an Interactor

and with prefixed I are internal events. In the case of the interactor specification, the simulation tree has 80 nodes and 239 events.

The automatic demonstration of properties expressed in action-based temporal logic (de Nicola et al., 1991) can also be obtained from the LOTOS specification of interactors. For example, we asked our tool if it is always true that an interactor provides an echo (**eo** event), by the feedback process, after receiving input data from lower levels (**im** event). The question was formulated by providing as input to the automatic tool, the LOTOS specification of the interactor and formulæ from temporal logic. The answer was positive.

Now we consider full LOTOS for the specification of a general description of a graphical interaction. This means that we can associate each process with a state and we can specify the data flow and the data types that are exchanged. The state of the collection process is a collection, i.e. a list of entities received from the outside. Some operations may be performed on the collection data type: **interpret**, adds entities to the collection or clears it; **trav_meas** and **trav_feed** interpret the output primitives in the collection and generate primitives at a lower level of abstraction: the former generates primitives that enable the measure to build its function, the latter generates primitives that have to be visualized by the feedback. Both functions are executed after another process has generated the control signal (the trigger) for activating them. Triggers are Boolean events that control the behaviour of the graphical objects.

```
process COLLECTION[ot, oc, cf, cm] (c:Collection) : noexit:=
    (oc?op:entity;
     COLLECTION[ot, oc, cf, cm](interpret(c, op))
    [] ot!true; cm!trav_meas(c); cf!trav_feed(c);
     COLLECTION[ot, oc, cf, cm](c)
endproc
```

The data type collection can be described as a general type. If we consider concrete interactions we have to specify the **trav_meas** and the **trav_feed** operations in addition to the entities.

```
type COLLECTION is LIST_ENTITY
sorts Collection
opns empty_col:                    -> Collection
     interpret: Collection Entity  -> Collection
     trav_meas: Collection         -> List_entity
     trav_feed: Collection         -> List_entity
     add:       Collection, Entity -> Collection
     clear:     Collection         -> Collection
eqns forall c: Collection, e: Entity ofsort Collection
     inq(e) eq clr =>
     interpret(c, e) = clear(c)
     inq(e) ne clr =>
     interpret(c, e) = add(c,e)
     clear(c) = empty_col
endtype
```

The data type Entity is defined in the following, where all the types of graphics primitives are indicated in the definition of the **Primitive_type**.

```
type ENTITY is
sorts Entity
opns mk_entity:   Primitive_type,Geometry,List_attributes -> Entity
     mk_entity_1: Geometry                                -> Entity
     mk_entity_2: List_attributes                         -> Entity
     inq:         Entity                                  -> Primitive_type
endtype
```

The state of the feedback process is a picture, which is a list of entities. Logically, it has the same description of the objects stored in the collection but at a lower description level from the graphics point of view. It receives primitives that compose the current picture and when an input data is received from the measure, it is used to detect primitives which are highlighted by changing related attributes, such as colour, in order to provide echoing to the user of the current primitives selected by the input device.

```
process FEEDBACK[cf, me, eo](p:Picture) : noexit :=
    (cf?p1: List_entity; me?id:Input_data;
      eo!highlight(pick(mk_pict(p1), id));
      FEEDBACK[cf, me, eo] (mk_pict(p1))
    [] me?id:input_data; eo!highlight(pick(p, id));
      FEEDBACK[cf, me, eo](p))
endproc
```

The data type picture is

```
type PICTURE is
sorts Picture
opns empty_pic:                -> Picture
     mk_pict: List_entity      -> Picture
     pick: Picture, Input_data -> Entity
     highlight: Entity         -> Entity
endtype
```

The measure receives entities from the collection. These are used to build its function, which must be applied to the input data. When the related trigger is verified, the current value (a higher level input data, built by applying the measure function on the received input data) is passed to the control.

```
process MEASURE[im, it, cm, me, md] (inp: input_data, p:List_entity): noexit :=
    (cm?p2:List_entity; me!meas(inp, p2);
      MEASURE[im, it, cm, me, md] (inp, p2)
    [] im?inp1:input_data; me!meas(inp1, p);
      MEASURE[im, it, cm, me, md] (inp1, p)
    [] it!true; md!meas(inp, p);
      MEASURE[im, it, cm, me, md] (inp, p))
endproc
```

The behaviour of the control function is simple. It just waits for a new value from the measure and then passes it to the outside.

```
process CONTROL[od, md] : noexit :=
    md?in:Input_data; od!in; CONTROL[od, md]
endproc
```

6. Examples of Graphical Interaction Specifications

To clarify how an interactor works, we briefly describe two common graphical interactions: a menu, a scrollbar and the movement of an icon within the proposed approach. All the interactors have the same dynamic behaviour. The main difference between a concrete example specification and the general one is in the definition of the specific collection and picture (the basic data types that they use and the related operations) which are both specializations of the general model. For this reason we focus our attention on the collection and picture specification in the next examples.

6.1. The Menu Example

In the case of the menu, the collection stores the output primitives and the identifiers of the elements of the menu. In the menu case each of them is associated with a rectangle, a string, a foreground colour, a background colour. When the output trigger is verified, it sends all the information to the feedback (by applying the **travfeedmenu** function) and the rectangles and the identifiers to the measure (by applying the **travmeasmenu** function). The measure function receives the position of the pointing device as input from the lower level and returns the identifier of the element, whose rectangle contains that position. This identifier is passed to the feedback, which realizes echoing by inverting the background and foreground colours of the related rectangle. This is performed by the highlight operation in the picture, which is applied to the element identified by the pick operation. When the input trigger is verified, the measure gives the identifier of the current selected element to the control which delivers it to the outside.

```
type COLLECTIONMENU is LISTENTITYMENU, LISTMEASENTITYMENU, LISTFEEDENTITYMENU
sorts Collectionmenu
opns  emptymenu:                      -> Collectionmenu
      interpretmenu: Listentitymenu   -> Collectionmenu
      travmeasmenu:  Listentitymenu   -> Listmeasentitymenu
      travfeedmenu:  Listentitymenu   -> Listfeedentitymenu
eqns forall id: Identifier, rec: Rectangle, st: String, fg, bg: Colour
     ofsort Listmeasentitymenu
      travmeasmenu(mkentitymenu(id, rec, st, fg, bg)) =
       mkmeasentitymenu(id, rec)
     ofsort Listentitymenu
      travfeedmenu(mkentitymenu(id, rec, st, fg, bg)) =
       mkfeedentitymenu(id, rec, st, fg, bg)
endtype

type PICTUREMENU is FEEDENTITYMENU, IDENTIFIER
sorts Picturemenu
opns emptypic:                                  -> Picturemenu
     add:        Feedentitymenu, Picturemenu -> Picturemenu
     mkpictmenu: Listfeedentitymenu          -> Picturemenu
     pickmenu:   Picturemenu, Identifier     -> Feedentitymenu
     highlight:  Feedentitymenu              -> Feedentitymenu
eqns forall id, id1: Identifier, rec: Rectangle, st: String,
          fg,bg: Colour, pic: Picturemenu
     ofsort Feedentitymenu
     highlight(mkfeedentmenu(id, rec, st, fg, bg)) =
      mkfeedentmenu(id, rec, st, bg, fg)
     id1 eq id =>
     pick(add(mkfeedentmenu(id, rec, st, fg, bg), pic), id1) =
      mkfeedentmenu(id, rec, st, fg, bg)
     id1 ne id =>
     pick(add(mkfeedentmenu(id, rec, st, fg, bg), pic), id1) =
      pick(pic, id1)
endtype
```

6.2. The Scrollbar Example

In the case of the scrollbar, the collection sends a rectangle primitive (with a fixed shape) and the related colour to the feedback for visualizing the bar. The measure, unlike the menu, does not need information from the collection. When the button is pressed the

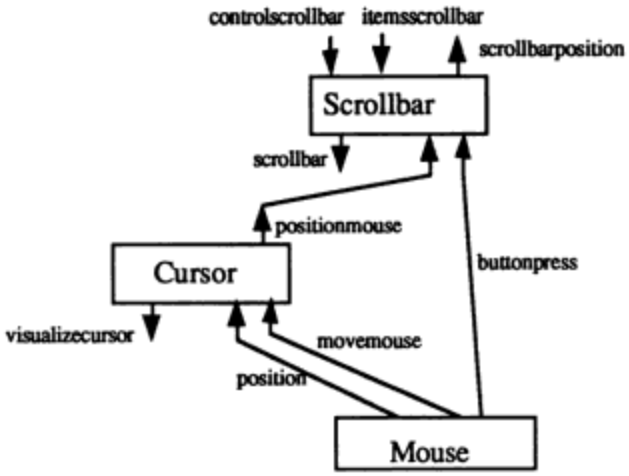

Figure 4: An Interaction with a Scrollbar

input trigger is generated. This means that for the input side, the current bar position is delivered to the control. For the output side, the rectangle related to the bar is placed in the new current position as indicated by the measure. If the user moves the cursor while the button is pressed, the bar is also moved. The bar movement is carried out by the highlight operation.

```
type COLLECTIONSCROLLBAR is ENTITYSCROLLBAR, FEEDENTITYSCROLLBAR
sorts Collectionscrollbar
opns emptycol:                                          -> Collectionscrollbar
     interpret: Collectionscrollbar, Entityscrollbar -> Collectionscrollbar
     travfeedscrollbar: Collectionscrollbar             -> Feedentityscrollbar
eqns forall cl: Colour, rct: Rectangle, coll: Collectionscrollbar
     ofsort Feedentityscrollbar
     travfeedscrollbar(interpret(coll, mkentityscrollbar(rct, cl))) =
      mkfeedentityscrollbar(rct, cl);
endtype

type PICTURESCROLLBAR is  POINT, COLLECTIONSCROLLBAR, FEEDENTITYSCROLLBAR
sorts Picturescrollbar
opns  emptypicscrollbar:                  -> Picturescrollbar
      mkpict: Feedentityscrollbar, Point  -> Picturescrollbar
      highlight: Picturescrollbar, Point  -> Picturescrollbar
eqns  forall cl:Colour, r:Rectangle, p1,p2:Point, coll: Collectionscrollbar
      ofsort  Picturescrollbar
      highlight(mkpict(mkfeedentscrollbar(r, cl), p1), p2) =
       mkpict(mkfeedentscrollbar(r, cl), p2);
endtype
```

In order to provide a complete specification of the interaction with a scrollbar we have to define an interactor associated to the cursor and a process with the mouse device (Figure 4).

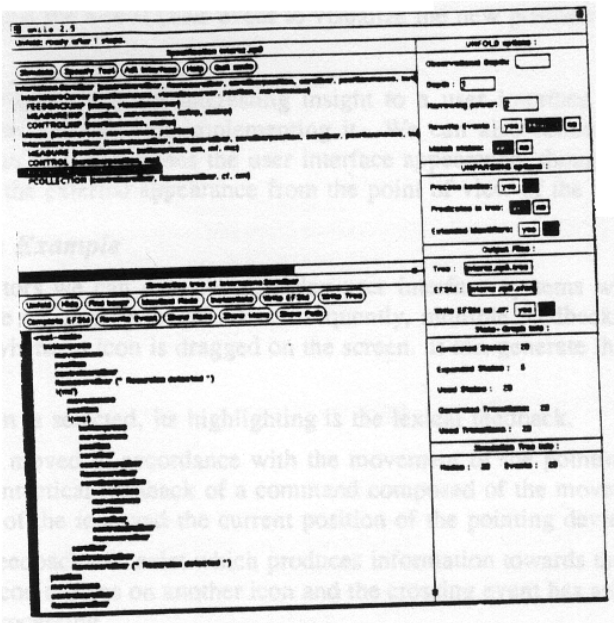

Figure 5: An Example of an Interactive Analysis of the Dynamic Behaviour of an Interaction

To make a scrollbar available to the user, an input interactor associated with the cursor is also needed. His trigger is the event associated with mouse movement. This interactor delivers the current position of the mouse and it has the cursor like an echo. There is also a process associated with the mouse which is only a physical device that can generate events to indicate the current mouse position and to indicate when the mouse has been moved or the button has been pressed. The following expression defines the composition among these entities.

```
(((InteractorCursor[positionmouse, visualizemouse, position, movemouse]
  |[positionmouse]|
InteractorScrollbar[controlscrollbar, itemsscrollbar, scrollbarposition,
                    scrollbar, positionmouse, buttonpress]))
  |[buttonpress, movemouse, position]|
(Mouse[buttonpress, movemouse, position]))
```

Figure 5 shows an investigation into the dynamic behaviour of a user interface composed of the mouse device and two interactors (scrollbar and cursor). The simulator was applied to the corresponding basic LOTOS specification, obtained by automatically projecting the FULL LOTOS specification. When the root node is unfolded, the five possible events appear (**buttonpress**, **movemouse**, **position**, **controlscrollbar**, **itemsscrollbar**). After selecting the **buttonpress** event and the unfold command, the list of the possible events that can be verified (after the **buttonpress** event) appears. We select the **mf** event (an internal event as the I indicates), performed in the scrollbar interactor to communicate

to its feedback process the new position where the bar should be visualized. This position is then communicated to the control by the **mf** event, and in turn by the **scrollbarposition** event to the outside. Afterwards, we are in a situation where the first set of five events is available with in addition the **scrollbar** event to visualize the new position of the bar to the outside.

This type of processing provides an interesting insight to a user interface system. We can understand its behaviour without implementing it. We can also follow the internal behaviour of the system that determines the user interface appearance, though this cannot be done by observing the external appearance from the point of view of the user.

6.3. The Move Icon Example

By composing interactors we can describe complex user interface systems with multiple parallel paths, multiple processing levels and consequently, multiple feedback levels. We consider an example where an icon is dragged on the screen. It can generate three possible feedbacks:

- When the icon is selected, its highlighting is the lexical feedback.
- If the icon is moved in accordance with the movement of the pointing device, it provides a syntactical feedback of a command composed of the move instruction, the identifier of the icon and the current position of the pointing device.
- A semantic feedback can exist which produces information towards the user when the selected icon crosses on another icon and the crossing event has associated any application processing.

To avoid obtaining a specification with several processes, we focus on how to performs the first two.

There are different ways to describe with interactors this typical graphical interaction. Here we describe it with four interacting objects: the interactor icons, with the definition of the icons that appear on the screen and its measure communicates the current selected icons (they may be more than one) and the related position; the interactor commands, is associated with a menu that allows the user to indicate the operation to be executed on the selected icon (copy, delete, move, resize). The result of the first object is transmitted, with the position of the mouse interactor, to the measure of the Instruction interactor. If a move operation has been selected in the commands interactor, it generates a trigger event for the Instruction interactor. It works out a set transformation primitive, whose coefficients are calculated according to the new position of the mouse and the current position of the selected icons. This entity is passed on to the icons interactor which then applies the transformation and visualizes the icon in the updated position.

We describe in greater detail each interactor. The icons interactor has in the collection the description of the objects visualized on the screen. Its output part performs the visualization. By triggering the button press event the measure function computes which is the current selected element(s) and transmits its(their) identifier to the feedback which provides echoing by inverting foreground and background colours and if, the trigger is verified, to the Instruction interactor. If more icons are selected the latter communicates them to an application process that realizes a specific processing depending on the selected icons. This processing can also generate a semantic feedback.

The icon type entity is similar to the menu. The difference is that each element, instead of being described by a rectangle, a foreground and a background colour, and a string is described by a list of points which define the icon geometry (it is fixed and cannot be resized), the icon colour and the icon border colour.

```
type COLLECTIONICONS is LISTENTITYICONS, LISTMEASENTITYICONS
sorts Collectionicons
opns travmeasicons:   Listentityicons  -> Listmeasentityicons
     trav_feed_icons: Listentityicons  -> Listfeedentityicons
eqns forall id: Identifier, p: Point, lps: Listpoints, cl, borcl: Colour
     ofsort Listmeasentityicons
     travmeasicons(mkentityicons(id, p, lps, cl, borcl)) =
      mkmeasentityicons(borcl, cl)
     ofsort Listenityicons
     travfeedicons(mkentityicons(id, p, lps, cl, borcl)) =
      mkfeedentityicons(id, p, lps, cl, borcl)
endtype

type PICTUREICONS is FEEDENTITYICONS, LISTFEEDENTITYICONS, IDENTIFIER
sorts Pictureicons
opns emptypic:                          -> Pictureicons
     add: Feedentityicons, Pictureicons -> Pictureicons
     mkpicticons: Listfeedentityicons   -> Pictureicons
     pickic: Pictureicons, Identifier   -> Feedentityicons
     highlight: Feedentityicons         -> Feedentityicons
eqns forall id, id1: Identifier, p: Point, ls:Listpoints, cl, bcl: Colour,
            pic: Pictureicons
     ofsort  Feedentityicons
     highlight(mkfeedenticons(id, p, ls, cl, bcl)) =
      mkfeedenticons(id, p, ls, bg, fg)
     id1 eq id =>
     pickic(add(mkfeedenticons(id, p, ls, cl, bcl), pic), id1) =
      mkfeedenticons(id, p, ls, cl, bcl)
     id1 ne id =>
     pickic(add(mkfeedentityicons(id, p, ls, cl, bcl), pic), id1) =
      pickic(pic, id1)
endtype
```

Commands is an instance of an interactor performing a menu. Its collection contains the output primitives that describe the menu (by drawing the rectangles and the strings inside them). In this case, too the measure receives the position of the mouse and the trigger event is generated by the button pressing. The feedback performs echoing by changing the background colour of the rectangle associated with the element identified by the measure.

The example is performed by the following expression:

```
(((device_mouse[position, movemouse, buttonpress]
  |[buttonpress]|
Int_Commands[control_menu, items,  move, menu, positionmouse, buttonpress])
  |[positionmouse, position, movemouse]|
Int_cursor[positionmouse, visualizecursor, position, movemouse])
  |[positionmouse, buttonpress]|
Int_icons[entity, entity, selectedicons, icons, positionmouse, buttonpress])
  |[move, positionmouse, selectedicons, entity]|
Int_Instruction[intersection, entity, selectedicons, positionmouse, move]
```

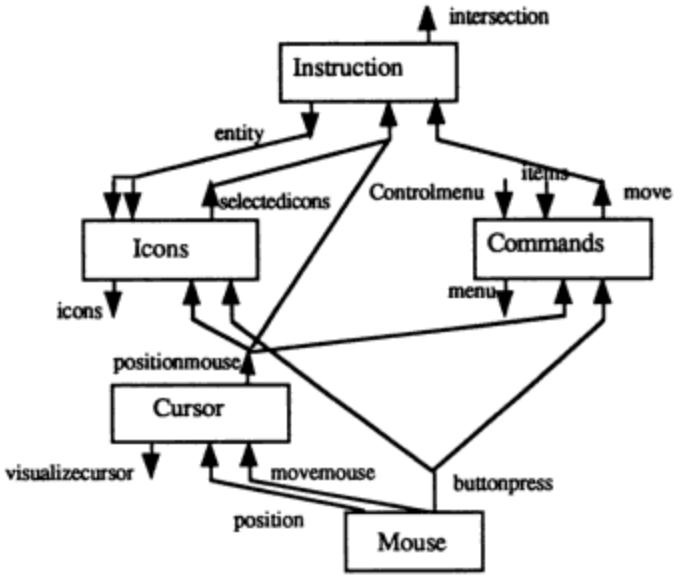

Figure 6: The Composition of Interactors Describing the "move icon" example

7. Conclusions

This paper has described an approach to the LOTOS specification of the design of UIS, composed of interacting graphical objects. We developed a structured and flexible design for this type of systems. By exploiting the LOTOS features and the available tools, we can obtain a complete specification that describes the state and events of each interacting object. After describing the abstract model of graphical interaction, some examples of common graphics interactions were discussed and first results of application of automatic tools in order to investigate their behaviour are presented.

This approach provides a sound ground for developing studies on UIS properties (Harrison, 1992; Paterno' & Faconti, 1991). We plan to define them by applying action-based temporal logic to the labelled transition systems associated with the LOTOS specifications. Further future work will extend the LOTOS specification to control objects and enable dynamic instantiation of interactors and their compositions in order to obtain more precise and configurable descriptions of UIS.

References

G D Abowd (1990), "Agents: Communicating Interactive Processes", in *Human–Computer Interaction — INTERACT'90*, D Diaper, D Gilmore, G Cockton & B Shackel [eds.], Elsevier Science (North Holland), pp.143–148.

D B Arnold, D A Duce & G J Reynolds (1987), "An Approach to the Formal Specification of Configurable Models of Graphics Systems", in *Proceedings of Eurographics'87*, pp.439–463.

L Bass et al. (1991), "The Arch Model: Seeheim Revisited", in *Proceedings of the CHI UIMS Developers' Workshop*, New Orleans, 28-29 April.

T Bolognesi & H Brinskma (1987), "Introduction to the ISO Specification Language LOTOS", *Computer Networks and ISDN Systems* 14, pp.25–59.

M Caneve & E Salvatori [eds.] (1991), *Lite User Manual*, Lotosphere Consortium.

U H Chi (1985), "Formal Specification of User Interfaces: A Comparison and Evaluation of Four Axiomatic Approaches", in *IEEE Transactions on Software Engineering* #SE-11 #8, pp.671–685.

J Coutaz (1987), "PAC: An Object-oriented Model for Dialog Design", in *Proceedings of INTERACT'87 — Second IFIP Conference on Human–Computer Interaction*, H J Bullinger & B Shackel [eds.], Elsevier Science (North Holland), pp.431–436.

D Duce, F Hopgood, R Gomez & J Lee [eds.] (1991), *Report of the Concepts, Methods and Methodologies Working Group in User Interface Management and Design*, Springer–Verlag.

D A Duce & L B Damnjanovic (1992), "Formal Specification in the Revision of GKS: An Illustrative Example", *Computer Graphics Forum* 11 (1).

D A Duce, P J W ten Hagen & R van Liere (1990), "An Approach to Hierarchical Input Devices", *Computer Graphics Forum* 9 (1), pp.15–26.

D A Duce, L S Marshall & E V C Fielding (1988), "Formal Specification of a Small Example Based on GKS", *ACM Transactions on Graphics* 7 (3), pp.180–197.

G Faconti & F Paterno' (1990), "An Approach to the Formal Specification of the Components of an Interaction", in *Proceedings of Eurographics'90*, pp.481–494, Montreaux.

G Faconti & F Paterno' (1992), "Specification and Verification of Graphical I/O Objects through the Temporal Logic Formalism", (To appear in Computer Graphics Forum).

A Goldberg & D Robson (1983), *Smalltalk-80: The Language and its Implementation*, Addison Wesley.

M D Harrison (1992), "Modelling Properties of the Interactive Behaviour of Graphics Systems", in *Formal Methods in Computer Graphics*, D A Duce [ed.], (In press).

ISO/IEC DIS 11 072 (1991), *Information Processing Systems, Computer Graphics, Computer Graphics Reference Model*, ISO Central Secretariat, Geneva.