The Visual Computer

A visual environment to define composition of interacting graphical objects

G.P. Faconti and F. Paternó

CNUCE – C.N.R., Via S. Maria, 36,
I-56100 Pisa, Italy

This work presents the FP visual language that specifies the components of a user interface and their relationship. Each component is an instance of an interactor that is a general description of a basic graphical interaction. By a visual language, it is possible to specify in a flexible way the logical structure of a user interface defined as a composition of interacting graphical objects. The graphical tool allows the designer to investigate the correctness of user interfaces and their properties.

Key words: Graphical interaction – User interface management systems – Visual languages

Correspondence to: F. Paternó

1 Introduction

A user interface management system (UIMS) (Duce et al. 1991) is comprised of a run-time support system and a set of design, specification, and evaluation tools. New trends in designing the architectures of UIMS (Duce et al. 1991; Bass and Coutaz 1991) focus on the description of sets of autonomous cooperating components exchanging input/output data and control information. The practical consequence of this approach results in the management of a very large number of logical modules working concurrently and communicating through a network that may quickly become very complex. Consequently, the logical structure of such an environment is more complicated with respect to traditional user interfaces.

Programmers have shown difficulties in specifying their application exploiting the capabilities of parallel languages mainly due to their custom of reasoning and working in a sequential way using sequential systems. One important problem is that in a textual parallel language the relationship across the modules realized by the concurrent commands are difficult to understand and to remember. This implies that the user interface designer has difficulties in obtaining correct specifications and investigating their properties, especially when using a parallel approach.

A new generation of visual languages (Shu 1985), based on the direct manipulation of multidimensional objects for building programming constructs has been proven to be of great help for specifying complex environments. These languages allow for an application to be graphically described, obtaining a specification that is easier to realize and intuitively clearer. Using visual languages, the productivity of programmers and users is dramatically increased with respect to the use of traditional methods. By directly manipulating graphical objects and specifying their relationship in a bidimensional space, it is possible to obtain representations closer to their conceptual model.

Chang (1986) distinguished the visual languages in four categories: *iconic languages for visual programming* using a visual representation for language constructs and for objects not inherently visual; *iconic languages to process visual information* that appear to the user as graphical representations for objects such as images; *textual languages to support visual interaction*, that make it possible to realize graphical interactions (such as GKS, PHIGS, PostScript and Xlib; *textual languages to process visual information* that manipulate images

The Visual Computer (1992) 9:73–83

and similar data common in the field of image processing, office automation, and robotics.
Following Chang's categorization, the aim of our work is to define a graphical language for specifying the logical structure of user interfaces that are not inherently graphic objects. Within this framework, the graphical specification of the logical structure of a user interface is addressed by a visual language for describing the instances of interaction object or interactors (Faconti and Paternó 1989) and their relationships.
A different taxonomy was also proposed by Myers (1990). He divides general programming languages depending on different criteria: languages based on visual programming (based on two or more dimensional specifications) or not on example-based programming or not interpretive or compiled. Visual programming is divided into two categories: 1) visual programming systems groupded depending on how they represent the programs (using flowcharts, Petri nets, data flow graphs, matrices, forms, and iconic sentences) and 2) program visualization systems (where graphics are used to illustrate some aspect of the program after it is written) classified depending on the constituents of programs they visualize, such as code, data, or underlying algorithmical structure. Here our system can be categorized under visual programming systems using data flow graphs as graphical representations.
One of the most commonly used techniques in graphically specifying user interfaces is in editing state transition diagrams. As an example, Jacob (1986) uses this technique to describe both the syntactical and lexical entities in separated windows. A more direct approach is found in ConMan (1988). Here the programmer realizes different types of interaction, manipulation, and visualization of graphical objects by requesting the services of a connection manager that allows for dynamic connections between the components. However, there is not a uniform description of the architcture of such components. In Singh (1990), a visual environment that mainly creates instances of interaction techniques and interactively refines their appearance is proposed, but the problem of creating higher-level interfaces by composing lower-level objects is not addressed.

2 Interactor model

In our visual environment we have graphic objects, namely icons, whose meaning is to represent other objects that are interactors; the difference is that the former ones are passive graphic representations (used in the specification phase) of the latter ones that have, other than the appearance (realized at run-time), also a dynamic behavior. A behavior of an interactor is further divided in two well-defined parts:
– *The external one*, which has an influence on the modification of the external appearance towards the user.
– *The internal one*, consisting of receiving, elaborating, and sending higher-level input data to other interactors or application processes.
In this way, we integrate the approach of graphics systems, such as GKS and PHIGS, and the approach of window-management-system toolkits. In the first ones, the logical input devices are characterized only by the data type they return (locator, pick and valuator) and the application programmer cannot affect their external appearance. While in the latter the interaction techniques are modeled in a class tree depending on their external behavior. They allow the designer to obtain user interfaces that are a flat distribution of interaction techniques and that interact immediately with application procedures. In fact, they can communicate among themselves only for layout management and exchange of basic events of the underlying window system, but they cannot be composed to realize hierarchical input data processing.
Visual language allows the specification of the logical structure of user interfaces supporting a large set of functionalities:

– *Unique description of objects from physical devices to applications.*
– *Generation of continuous and multiple feedback from within the user interface.*
– *Capability of handling multiple threads of interaction simultaneously.*
– *Extendability of functionality.*
– *Various ways to control the dialog between user interface and application.*

To obtain these goals, user interfaces are logically developed by composition of interactors. One interactor is an entity supporting the conversion between higher- and lower-level input and/or output in both directions. It has the aim of providing a general description of an interaction with graphical devices. Another attempt to encapsulate dynamic behavior of graphical interaction is in Myers

(1989), where a set of possible general descriptions are indicated while we are able to provide a unique abstract description for all possible interactions with graphical devices. In Duce et al. (1990), the hierarchical composition of only input devices is addressed.

An interactor is seen by itself as a set of concurrent processes describing one basic graphical interaction. The capabilities of interactors have been described by using the typical constructs provided by concurrent languages to manage the exchange of messages, activate and deactivate processes, evaluate concurrent situations, and handle parallelism in general. This allows us to specify the cooperative tasks between the user interface components in a very flexible way.

There have already been some proposals for appliying concurrent languages to the user interface specification (Cardelli and Pike 1985; Hill 1986). One strong problem with using them is that the textual syntax of those languages has generally hidden meanings and consequently the resulting programs are not easily understood by non specialists. For this reason, it is obviously important to address the capabilities offered by visual programming (because they are more clear, more powerful, and the interrelations among different items are expressed in amore understandable way) to specify concurrent interactors and their behavior.

A basic interaction unit, or interactor, mainly consists of five functions:

- *Trigger*, to identify a significant moment in time.
- *Measure*, applied to the input data (when the trigger is verified) to build a new input data for the next higher level.
- *Feedback*, to generate the output toward the user (a picture composed of lower-level output primitives).
- *Collection*, to describe the output primitives associated with the interactor that are interpreted and sent to the feedback for visualization.
- *Control*, to deliver the result of the measure to other interactors or to the application.

Each of these functionalities is distinguished by a set of parameters and all together make it possible to encapsulate the general behavior of a basic graphical interaction. A visual language for the graphical specification of the parameters indicating instances of interactors and their possible connections is a powerful tool that gives the designer the capability to develop the user interface in more easily. On the resulting specification, automatic tools can be applied to investigate properties and dynamic behavior.

3 What the language has to specify

The model of an *interactor* was described through constructs of the ECSP concurrent language (Baiardi and Vanneschi 1985), which is an extension of CSP (Hoare 1985) already used as specification language for graphical input (Duce et al. 1990). CSP is a formal notation to describe concurrent processes that communicate by synchronous events. ECSP provides, more than CSP, explicit and compact ways to use different protocols: symmetric synchronous (where two components communicate in a synchronous way), asymmetric synchronous (more processes can send data to one process in a synchronous way) and symmetric asynchronous (two processes communicate in an asynchronous way by a buffer) and has some dynamic features, such as the capability of allocating communications channels by means of variables of type process name. These allow a process to use the same channel to receive data from different processes at different times. The channels are modifiable data that belong to the receiving process. It is possible to change the sender process part.

An interactor can belong to three different categories (input/output, input, output) depending on if it has an external and internal behavior, only the internal one, or only the external one. In the more general case (input/output), it can communicate with the outside by six channels (four input and two output channels) as presented in Fig. 1. It receives input in four channels: one to receive control requests, one to receive the high-level output primitives defining the external appearance of the inter-

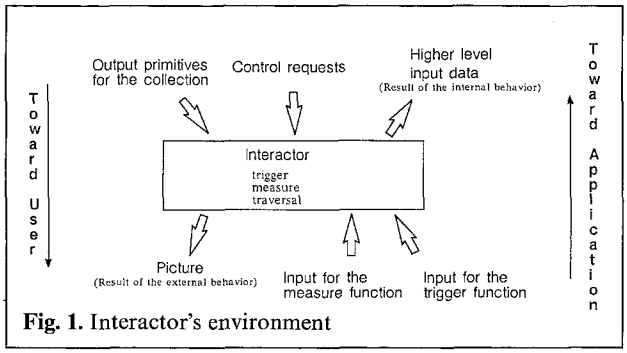


Fig. 1. Interactor's environment

actor into a collection, one to receive input to the trigger function, and one to receive input to the measure function. It generates output by two channels: one to send the high level input data that it produces to other interactors or application processes (this defines its internal behaviour) and one to produce output information after having processed the primitives internally stored in a composition (this defines its external behavior).

Our graphical language is based on the ECSP specification of interactors: it provides a graphical representation of constructs built on top of the language. Another choice was possible: to define a graphical syntax of ECSP to describe interactors behavior and their composition. In this way, we would have obtained a very large, difficult-to-interpret, graphical representation. We chose to associate a compact graphical representation directly with the set of modules of ECSP commands describing an interactor. This allows us to obtain clear and compact graphical representations and then we can further define the particular behavior of an interactor instance by interactively providing the current value of a set of parameters.

The parameters required to completely define a user interface based on interactors are of two types (in Sects. 5 and 6 we describe how the designer can specify them):

– *To indicate specific elaborations realized by each interactor* – data types transmitted on the channels and functions to define processing of the trigger and the measure functions.

– *To define their cooperation* – sender and the receiver for each external channel, which means to define the connections among interactors in order to realize their composition, obtain complex interactions and provide the logical structure of the entire user interface (defining all the possible paths for the data exchanged between user and application).

Composition rules among interactors allow them to send the result of the internal behavior of one interactor as input for the functions defining the internal behavior of another interactor. The result of the external behavior is as sent input to the function defining the external behavior of another interactor.

When an interactor has to be specified, the first step is to select the class defined by the data type it returns towards the application (internal behavior). Then its subclass is defined by indicating the data types it receives in order to elaborate its input

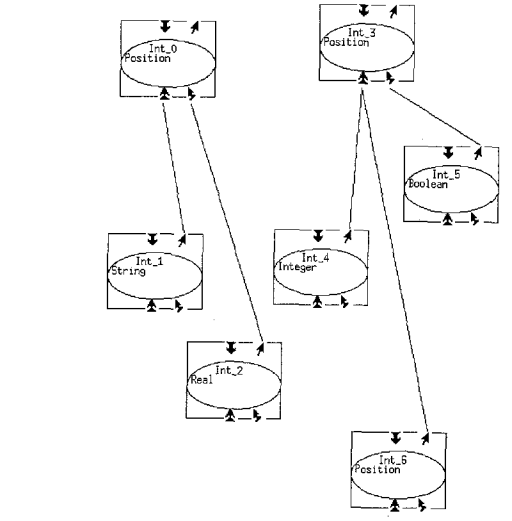


Fig. 2. Hierarchical composition of input interactions

functions. As an example, Fig. 2 shows a set of input interactors, where two interactors (Int_0 and Int_3) have the same class (Position), but with different subclass (Int_0 receives for the trigger a Real and for the measure a String; Int_3 receives for the trigger a Boolean and for the measure an Integer and a Position).

The physical devices can be seen as particular instances of interactors of input or output classes. Their distinguishing features are they can only generate events depending on user actions or visualize graphic output generated by other interactors. When the programmer specifies a user interface, he begins by indicating the physical devices that will be used by the user interface by placing the related icons in the low part of the space for the visual program. Then, going upwards, he has to indicate the interactors communicating with physical devices until arriving at the interactors that communicate with the application processes.

This type of logical structure implies a temporal ordering among the events generated by interactors, because one of them at a certain level does not generate its data towards the application until all the others in the paths between it and the physical interactors have generated their data. At each level of composition, it is possible to generate output feedback indicating the evolution of a complex interaction. When the user interface is specified in this way, its control is mainly determined by the connections along which the data flow and by the trigger functions that can be considered Boolean

functions. When they are verified, the related interactors generate higher-level input data for other interactors or the application.
ECSP provides dynamic channels. This allows us also to reconfigure dynamically the set of communications among the interactors. The only constraint is that the data produced are always of the same type. More generally speaking, it is possible to have a dynamic environment where the designer can change the current set of instances of interactors and their connections. For example, if we have interactor A returning a triplet defining a rotation in the 3D space, it can receive the three Reals from three Scrollbars, but a reconfiguration of its input-measure channel can be realized in order to receive the three Reals from interactors associated with three textual inputs by the keyboard.

4 Definition of the visual language

The syntax of a language defines how to compose its lexical elements. Textual programming languages have a syntax that is defined by particular rules applied on string elements that can be combined only in a linear sequence. In the case of visual languages, we have a graphical syntax defining the rules that have to be applied on graphical symbols that can be composed in various ways over the bidimensional space of the screen. To define our visual language for user interfaces based on interactors, the main items to specify are:
– *Instances of composing interactors.*
– *Connections among them,* in order to realize the logical structure of the user interface and completely define its behavior.
Definition of the graphic syntax provides the capability of recognizing in a precise way all the possible graphic representations associated with elements of the language. A more detailed discussion on visual-language specification is found in Paternó (1990). In general, there are two approaches (Fig. 3) to obtain a graphic syntax:
– *Direct* (Tortora and Leoncini 1988; Golin and Reiss 1989), where it is defined by adding to the usual grammar operators (concatenation and substitution) some topological operators that are applied to the terms of the language (for example, vertical concatenation and spatial overlay). In this approach, attribute grammars are often used to define completely the position of graphic symbols. These define the geometric attributes of non-terminals symbols using semantic functions associated with production rules.

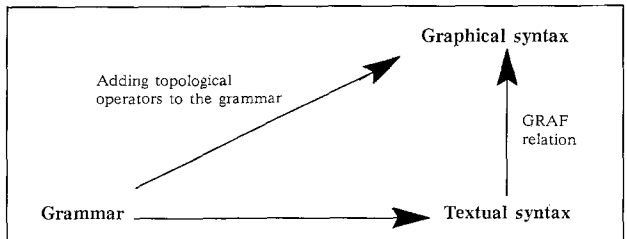


Fig. 3. Two different approaches to a graphic grammar

– *Indirect* (Bolognes and Latella 1989), where grammars using the usual operators are employed and a relation can be defined between the resulting language and the graphic language. In this case, we have a textual and a visual language, which can be seen as two concrete syntaxes having the same abstract syntax tree. This solution confines in the relation, for clearness and modularity, the problems related to the construction of the graphic representation.
We have chosen the second approach, which is to realize the language by using a traditional grammar, but with particular keywords. These are used by the relation (we have called it GRAF) to obtain the associated graphic representation.
In our case, the GRAF relation replaces the specific keywords with elementary graphic representations of fixed size (see Fig. 6) with strings, ovals, boxes. These graphical elements are composed by means of presentation operators in order to generate the graphic specification associated with the expression of the language. Otherwise, it is also possible from the graphic representation to identify the related textual expressions. This is important, because from the latter it is easier to obtain the equivalent expression of a language that can be executed by a running system. Figure 4 describes the general architecture of the environment that allows the designer to use FP language with the purpose to specify user interfaces based on interactors. The FP language has a graphical and a textual syntax. The designer can specify an expression of the language by a graphical editor. Then the editor, following the indications of the GRAF relation, allows locating the associated textual expression. At that moment, we have three possibilities: archive a file for other processing that can be realized at other moments, translate the specification in a programming language to obtain executable code,

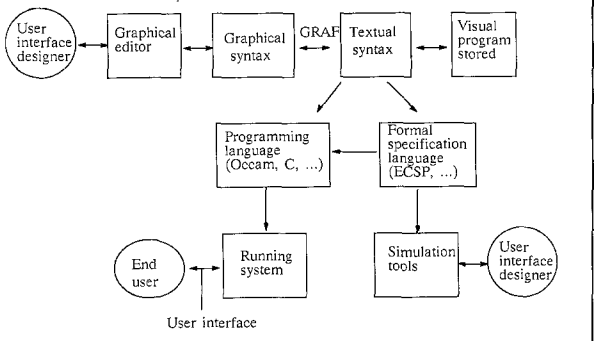


Fig. 4. Architecture of the operating environment

and translate the specification in a general-purpose formal notation (in this way, it is possible to apply on it tools to simulate the specification and investigate the associated properties). From the specification in a formal notation by a specific available tool, it is possible to translate directly in a programming language.

When the specification of a user interface is defined in a formal notation, it is possible to apply some tools, other than to verify its correctness, to investigate if properties, such as deadlock-free and equivalences, are verified and to simulate its dynamic behavior. Then the specification may be refined in a programming language to obtain the executable system. The external appearance of each interactor at run-time is defined by a set of default values or by primitives provided by the application. It can be modified during the execution by receiving new data from application processes or from other interactors into the related collection.

By defining different relations between the textual and the graphical syntax, it is possible to obtain different graphic representations for the same language. This provides the designer with one of many choices, depending on his requirements, skill, or tastes. For example, in our case it is possible to indicate the partner process over a channel by a pop-up menu, which visualizes its identifier at the click of the mouse on the channel representation or by a straight segment connecting the representation of the two communicating interactors. The first choice is better suited when there are too many graphic items and the set of lines is very complex and difficult to interpret.

The graphic specification is realized by a syntax-driven editor. It allows the designer to select interactor instances and their parameters choosing among a set of possibilities. For each type of choice, it is possible to define new possibilities. One of the advantages of a syntax-driven editor is that the designer is guided during the specification, and it is possible to immediately check for syntactical errors. For example, if it is specified that the value produced by an interactor has to be sent to another one and the latter does not have that data type among its input data types, an error will be signaled. Obviously, lexical errors are prevented by selecting an item by pointing instead of by typing.

The grammer rules in Fig. 5 show that in the FP language a user interface is seen as an identifier and a list of interactors. Each interactor has an identifier and a category. Depending on the category chosen, there are different definitions, because each category has a different set of channels to communicate with the outside. Once the communication channels are defined indicating the partner processes and associated data types, logical connections among interactors are also recognized and the structure of the user interface is finally specified.

In Fig. 6, some rules used by the relation are presented to expand the expressions of the graphic language in the graphic space. This is done by using some presentation operators (such as inside and

Graphical grammar rules to define a user interface

<user interface> = GUI (<interface_id>, <list_interactor>)

<list_interactor> = <interactor> | <interactor> <list_interactor>

<interactor> = GINT(<interactor_id>, <int_category_id>) | <device>

<int_category_id> = <input_category> | <output_category> | <input_output_category>

<input_category> = GINTI(<int_class>, <ch_as>, <ch_ss>, <ch_as>, <ch_as>)

<ouput_category> = GINTO(<int_class>, <ch_ss>, <ch_as>, <ch_ss>)

<input_output_category = GINTIO(<int_class>, <ch_ss>, <ch_as>, <ch_ss>, <ch_as>, <ch_as>, <ch_ss>)

<device> = <device_type> <interactor_id>

<device_type> = <input_device> | <ouput_device>

<input_device> = <physical_input_device> <ch_ss>

<output_device> = <phisical_output_device> <ch_as>

<physical_input_device> = MOUSE | KEYBOARD | ...

<physical_ouput_device> = SCREEN | ...

<ch_ss> = SA(<process_id>, <ch_data_type>, <process_id>)

<ch_as> = LA(<list_process_id>, <ch_data_type>, <process_id>)

<ch_sa> = QA(<process_id>, <ch_data_type>, <process_id>, <int_exp>)

<int_class> = Integer | Position | Real | String | Null | ...

<ch_data_type> = <output_primitive> | <control_request> | <input_primitive>| ..

Fig. 5. User interface grammar

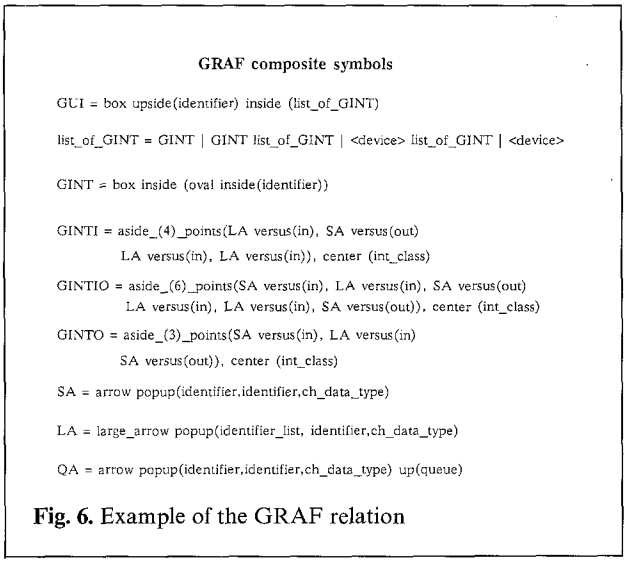


Fig. 6. Example of the GRAF relation

upside) applied on the alphabet of previously defined elementary graphics symbols with fixed size (such as box and oval). The operators indicate some constraints related to the position of symbols that have to be verified among the symbols of the graphic representation. For example, the representation of an interactor is composed of a box inside an oval, which contains the name of its identifier and its class. The representations of the channels are placed on some previously defined point, depending on the interactor category. They are differently shaped arrows, depending on the protocol used and on the direction of the data flow.

5 Graphical environment

After having defined the visual language, we have also realized a tool to provide a more friendly environment to specify the logical structure of user interfaces. We have added new functions other than the specification of a visual program. The functions supported are described below.

Modify the instances of interactors

When the designer begins his work session, he must indicate if he wants to create a new interface or modify one previously defined. Then he can work on windows divided into two areas: a Work Area containing the results of the designer choices, and a Command Area containing the graphical representations of the available categories of interactors and the indication of the set of commands to continue his work (Fig. 7). It is possible to create and destroy instances of interactors.

A better methodology for building a graphic environment to describe the structure of the user interface is to place in the lower side of the graphic space those interactors associated with physical devices, then those connected with them, and so on up to the higher side of the graphic space.

In creating one instance of interactor, the designer first indicates its category by selecting the associated graphical representation. Then the icon associated with the selected category (in the Command Area) is substituted by a sequence of menus allowing the designer to select the values for each interactor parameter: first, the class (data type produced by the interactor), then the subclass (set of input data types for trigger and for measure), and finally the trigger and measure functions defined by composing different kinds of available operators (logical, temporal, and numerical). The user-interface designer indicates them by pointing on a menu listing the representations of the available ones. At the moment of specifying the input function, the set of data types previously selected in defining subclass appear. The designer must indicate in which order to apply the selected operators to the selected data types. If the editor does not detect any mistake, such as class not corresponding to the data type delivered by the measure function, the function is stored in the definition of the interactor. A typical example of an input measure is a function that delivers an object identifier depending on the value of a position received as input.

Edit the connections

When this option is chosen, the editor asks a person to select in the Work Area the interactor whose connections have to be defined. After its selection, it asks the designer to indicate which interactors send input for the measure and trigger functions of the selected interactor and to which interactor to send the produced output picture. The designer can change the logical structure of the user interface changing the communication channels either because the set of existing interactors has changed or because he wants to modify the possible data paths. This is obtained by means of the capabilities of the underling support of ECSP that allows modi-

fying the sender processes of each channel. Connections are stored in the editor system so that successively related requests can be answered.

Visualize the connections

In normal operations, the connections are not graphically represented in order to avoid having too complex and difficult-to-understand representations of the user interface. At any time, it is possible to require each interactor to visualize the corresponding connections. This is realized by selecting the *View Connections* command. Then the editor asks the designer to select an interactor. A double button appears to allow the designer to indicate if the connections of the selected interactor should be visualized in a simple or complex way. In the first case, the lines connecting the graphic representation of the selected interactor with those communicating with it are shown. In the complex way, it is possible to visualize the entire graph representing the connections between the selected interactor and the others, and then, recursively, those of the connected interactors. In this way, it is possible to see all the interaction paths arriving to that interactor and leaving from the physical devices.

Save for further elaborations

It is possible to save the specification of a user interface to process it again later. The associated textual specification of the expression of the graphic language is saved with an identifier name indicated by the designer. When it is selected for further manipulation, it is interpreted by the GRAF relation to obtain again the associated graphic representation.

Visualize the current values of a interactor

You can select an interactor to visualize the current values of the parameters describing its behavior. These appear by pop-up menus.

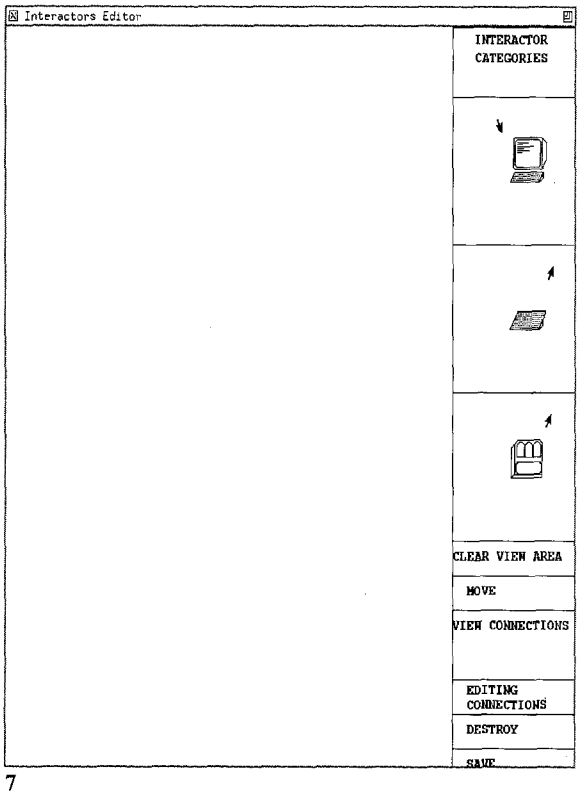
Select an activation or deactivation status

In complex user interfaces or in the prototypation phase, the capability to activate only part of the available interactors can be useful. In this way, only a subset of allocated instances can receive, elaborate, and send data. The deactivated interactor highlights their state to the user by blinking their graphic representation.

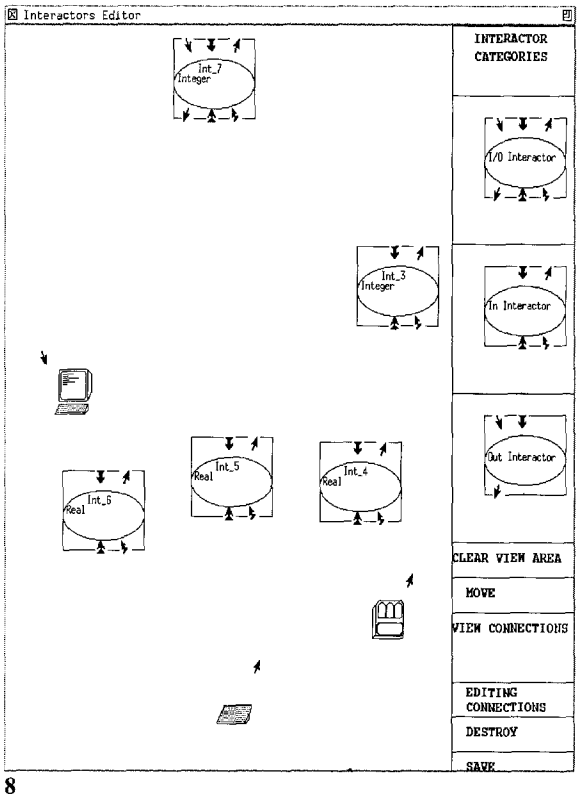
6 Example

In this example, the logical structure of a composed interaction is specified using eight interactors: six of input category, one of input/output category, and one of output category which are layered in three levels (one associated with the physical devices, one with the input interactors, and one with the I/O interactor). They are composed to realize the following composed interaction: the user can provide via keyboard three Reals (each one produced by the internal behavior of a specific interactor), allowing him to indicate a rotation in 3D. He can also select a visualized object via the mouse. The object identifier (an integer) and the three Reals can be sent to an I/O interactor that produces, as result of the external behavior, the selected object rotated by the specified value. The result is sent to the screen. The I/O interactor computes the rotated object when its trigger is verified. In our example, it receives input to the trigger from the interactor producing the object identifier. This means that each time it receives a new object identifier it produces its rotated representation, while each time it receives new rotation coefficients these are stored to be applied when the trigger is verified. If the three Reals also were sent as input to the trigger function of the I/O interactor, its behavior would be modified, because the trigger also would be verified for each new rotation coefficient. This would mean that the last-selected object is rotated each time a rotation coefficient is modified.

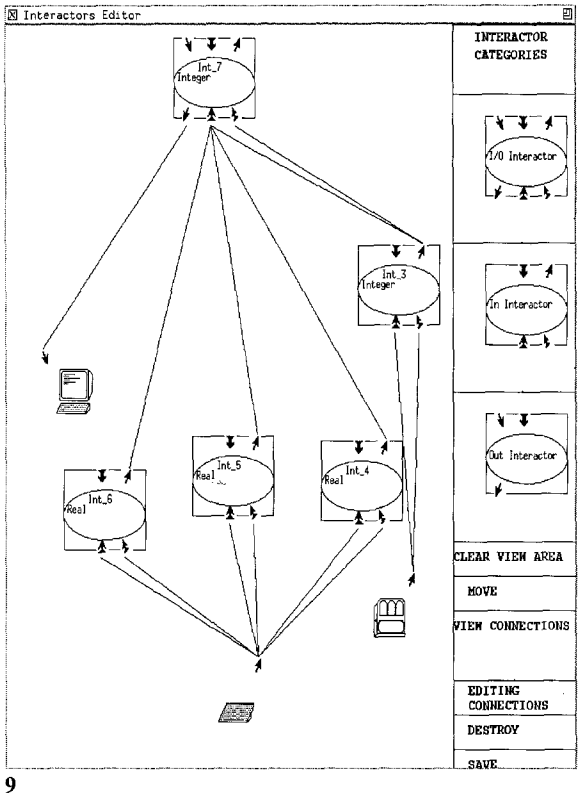
At the beginning, the designer starts the Interactors Editor, whose initial layout is shown in Fig. 7. It is divided into the Work Area and the Command Area with commands and the graphic representation of available interactors categories. At the beginning, the designer must indicate which physical devices his user interface will use. Then he starts to create instances of interactors. This means that he firstly selects the associated category and then a list of menus appears asking a person to indicate the input data types for the measure and trigger functions. Then the related processing has to be defined. The lists of available operators input data



7



8



9

Fig. 7. Initial layout of the Interactors Editor

Fig. 8. Instances of interactors

Fig. 9. Extended visualization of connections for an interactor

types previously selected appear. Then the designer defines the function by indicating how the operators he selects should be applied to the data types. Once the interactor instance is completely defined, its graphical representation (indicating category, class and identifier) can be placed in the Work Area.

After having defined the instances of interactors (Fig. 8) composing the User Interface System, the designer must indicate how they are connected. This means selecting the *Editing Connections* option. The editor asks a person to select an interactor instance and indicate from which interactor it receives input to the measure and trigger functions and, if it has also output functionality, to which interactor to send the resulting external behavior of the current interactor. After defining the connections, they are not immediately visualized. The designer can ask the editor to visualize or to make invisible the connections in a flexible way. In Fig. 9 all the connections belonging to communication paths passing through Interactor 7 are visualized:

7 Conclusions

The examined approach provides a way to define the logical structure of user interfaces based on interactors. The designer of the user interface can realize it in a simple way using direct-manipulation techniques. It is possible to check the presence of mistakes in the specification as soon as it is realized, because the tool knows the rules specifying the language and communicates the detected mistakes immediately to the designer. The result of the graphical specification is a compact representation of the interacting objects composing the user interface and their relationships. This can be very useful for the designer to quickly understand the processing accomplished by the user interface to realize its internal and external behavior.

Before refining the user interface specification in a programming language, it is possible to apply specific tools to investigate its properties (Paternó and Faconti 1991). For example, it is possible to check if all interactors are crossed by a path between user and application and if there are possible deadlock situations.

In this way, we have a visual language to develop unser interfaces that provides the designer with an extensible and flexible set of functions. This is very important, because it makes it easier to realize a user interface with the preferred interaction style. This approach provides software reusability, because it is easy to create libraries with typical complex interactions defined by a specific set of interactors. When these are useful, it is sufficient to connect the other interactors of the user interface with their external channels, without defining them again.

Future developments of this work are to make it possible to access and manipulate directly the data stored in each collection by dedicated tools that are already developed for particular environments (Howard 1990) and extend the visual language to indicate dynamic conditions for activating/deactivating interactors. We are moving the obtained specification from a set of modules implemented with the ECSP notation to a set of LOTOS modules, because for specifications realized in this formal notation more flexible and powerful tools are available to investigate the behavior of the realized specifications.

We also want a better implementation on a multiprocessor architecture. A user interface can be composed of numerous interactors. Each interactor consists of different functionalities. It is an autonomous entity that executes in parallel with respect to others. This higher architectural parallelism improves the performance of user interfaces if it is implemented on a parallel with respect to others. This higher architectural parallelism improves the performance of user interfaces if it is implemented on a parallel system closely related to the graphical hardware. Now they are implemented on a high-resolution workstation with a set of transputers on board where the interactors, implemented by Occam, can be executed.

References

Bajardi F, Vanneschi M (1985) Linguaggi per la programmazione concorrente. Franco Angeli Libri, Milan

Bass L, Coutaz J (1991) Developing software for the user interface. Addison Wesley

Bolognesi T, Latella D (1989) Techniques for the formal definition of the G-LOTOS syntax. Proc IEEE Workshop on Visual Languages

Cardelli L, Pike R (1985) Squeak: a language for communicating with mice. ACM Comput Graph 19(3):199–204

Chang SK (1986) Introduction: visual languages and iconic languages. In: Chang SK, Ichiwata T (eds) Visual languages. Plenum Press, Ligomenides, PA, pp 1–7

Duce DA, van Liere R, Ten Hagen PJW (1990) An approach to hierarchical input devices. Comput Graph Forum 9(1):15–26

Duce D, Hopgood F, Gomez R, Lee J (1991) User interface management and design. Springer Verlag, Berlin Heidelberg New York

Faconti GP, Paternó F (1989) An approach to the formal specification of the components of an interaction. Proc EUROGRAPHICS '90

Golin EJ, Reiss SP (1989) The specification of visual language syntax. Proc IEEE Workshop on Visual Languages

Haeberly PE (1988) ConMan: a visual programming language for interactive graphics. Comput Graph 22(4):103–111

Hill RH (1986) Supporting, concurrency, communication, and synchronization in human-computer interaction – the Sassafras UIMS. ACM Trans Graph 5(3):179–210

Hoare CAR (1985) Communicating sequential processes. Prentice-Hall International, London

Howard TLJ (1990) A structure network visualiser for PHIGS. Comput Graph Forum 9(2):139–147

Jacob RJK (1986) A visual programming environment for designing user interfaces. In: Chang SK, Ichiwata T (eds) Visual languages. Plenum Press, Ligomenides, PA, pp 87–107

Myers B (1990) Taxonomies of visual programming and program visualization. J Visual Programming and Languages 1(1):97–123

Myers B (1989) Encapsulating interactive behaviours. Proc CHI Conf '89, pp 117–123

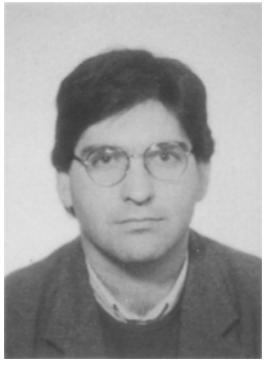
Paternó F (1990) A comparison of approaches to the formal specification of visual languages, CNUCE Internal report

Paternó F, Faconti GP (1991) Toward the definition of properties of user interface systems. CNUCE Internal report

Shu N (1985) Visual programming. Van Nostrand Reinhold

Singh G (1990) Vu: visual user-interface design. The Visual Computer 6(4):230–241

Tortora G, Leoncini P (1988) A model for the specification and interpretation of visual languages. Proc IEEE Workshop on Visual Languages



FABIO PATERNÓ received his Laurea degree in Computer Science at the University of Pisa in 1984. In 1986 he joined CNUCE where he worked in the design of graphics systems. In 1991 he was visiting scientist at the Rutherford Appleton Laboratory, Oxford, where he worked in the field of Formal Methods in Computer Graphics. His recent interests include Visual Languages, Formal Methods for Interactive Systems, Design of Graphical User Interface Systems. He is member of ACM, EG, AICA.

GIORGIO P. FACONTI is Senior Researcher at the National Research Council of Italy – Istituto CNUCE, where he presently acts as the head of the Department of Architecture of Software and Hardware Systems and as the CNR representtive for Computer Graphics and Human Computer Interaction to the European Research Consortium for Informatics and Mathematics (ERCIM). He obtained his M.S. in Mechanical Engineering and his Ph.D. in Computer Science from the University of Pisa respectively in 1971 and 1974. He has been involved in Computer Graphics since 1974 both in industries and research institutions. He has been involved in the process of standardization in Computer Graphics from 1984 and is actually the head of the Italian delegation at ISO IEC/JTC 1/SC 24, where he is involved in the Computer Graphics Reference Model Work Item. His main area of research is on Formal Specification and on Architectures for Interactive Systems, and on Visual Languages.