Volume 13, (1994) number 4 pp. 199–210

COMPUTER GRAPHICS forum

The Design and Specification of a Visual Language: An Example for Customising Geographic Information Systems Functionalities

F. Paterno'*, I. Campari*†, R. Scopigno*

* Istituto CNUCE, Consiglio Nazionale delle Ricerche, Via S. Maria 36, 56126 Pisa, Italy
† Department of Geoinformation, Technical University of Vienna, Gusshausstrasse 27-29 A-1040 Vienna, Austria

Abstract

In this paper the design of a visual program editor and its specification using formal grammars are discussed. We consider an environment to specify, analyse and execute visual programs for a Geographical Information System (GIS). The lack of sophisticated user interfaces is one of the major drawbacks to Geographical Information Systems, particularly for people without a sound background in computer science. The use of a visual language approach is useful in order to hide the plethora of basic GIS functions, while providing ready-to-use tools to solve users' tasks. The visual environment provides users with higher level interfaces; it is based on the module concept, which is conceived as a software building block that implements a solution to a general basic task and is presented to the user through an interactive frame. Complex GIS queries can be carried out by interconnecting modules into flow networks, using a direct manipulation approach.

1. Introduction

There is currently much interest in visual programming and visual languages[1] as they seem to be a powerful tool for people who are not experts in programming to access the functionalities of an application. There have been proposals to categorise these kinds of languages into different classes[2]: *visual environments*, by which not inherently visual information is presented to the user in a graphical form; *languages for handling visual information*, mainly designed for managing visual information or images; *languages for supporting visual interaction*, where icons and graphical objects are used as a means of communication with computers; and *visual programming languages*, allowing users to actually program with visual expressions.

A different taxonomy was also proposed by Myers[3]; it is based on the evaluation of different characteristics, e.g. languages based on visual programming, on example-based programming, interpreted or compiled. Analogously, he categorises visual programming systems depending on how they represent the programs (using flowcharts, Petri nets, graphs, matrices, forms, iconic sentences and so on). Program visualisation systems are classified depending on the components of the programs they visualise: code, data or the underlying algorithmical structure.

The main developments in the field of visual programming[4] fall into two classes. On the one hand, there is the creation of visual environments to specify and debug programs, more generally in the software engineering field. On the other hand, there is the evolution of languages to manipulate visual information, to support visual interactions and to program with visual expressions. In the latter field, there have already been proposals to use visual programming to specify user interfaces or to interact by visual expressions with systems of different types. In the computer graphics and visualisation field many systems have been proposed based on the use of a dataflow approach. ConMan[5] was designed as a connection manager which develops dynamic connections between predefined modules that are visually represented on the screen; each module supports different types of interactions and graphic visualisations. A similar approach was also used in the design of the successful AVS[6] and apE[7] visualisation systems. With the same philosophy, VILD was defined as a visual language for access to an object-oriented multi-media database system[8].

Visual representations of computational flow are

widely accepted and with current graphical technologies, effective and efficient iconic diagram editors and interpreters can be developed. Dataflow representation, based on the use of graphic symbols connected by arrows, is generally clear and immediate for the user: priorities and processing order between the represented items can be directly acquired from an analysis of the topology of the graph. Moreover, given a dataflow specification it is simple to detect the components that can be executed in parallel, thus making the design of parallel executions simpler. In Hils[9] there is a survey of data flow visual programming languages which indicates that these kinds of visual languages have been most successful when they have a narrow, fairly specialised application domain and when they are intended for use by non-programmers or novice programmers. Diagrams are only considered to be effective when they represent the relationships between a limited number of elements, in other words, when they give a high level representation of a procedural flow. This is a well-known limitation of the diagram as a means of representation.

One important problem is to identify formal notations which allow us to describe in a precise and unambiguous way which graphical representations belong to the visual language. The techniques which are used for textual languages are not sufficient to do this because instead of textual symbols ordered linearly we have graphical symbols which can be composed in the bidimensional space of the screen. This implies that we have a wide set of possibilities about how to compose them and we need notations which indicate the legal ways to compose them. In the paper we show a couple of possible approaches to the design and specification of a visual language but there are different possibilities: another example of a formal notation for visual languages is provided by relation grammars[10] which are an extension of textual grammars and which allow the explicit expression of relationships among the components of a two-dimensional structure.

Configurability, data-flow representation, the introduction of the user perspective in the software design, and the use of formal methods in the definition of the visual language all improve the design and the usability of visual software. In this work we discuss how these concepts are used to build a visual environment allowing users to interact with the functionalities of geographical information systems (GIS) by using a graphical environment.

GISs are information systems incorporating spatial databases. They have been developed to satisfy multiple applications in various disciplines all characterized by the common need to handle spatial data. The path to satisfy the user goal is strictly connected to the technical ability and professional background of the user[11]. Therefore there are as many methodological paths as types of users. This methodological dissimilarity characterises GIS users and also means that the constraints and the rules of the tools are conceived by users in a different way, even though the final goal may be identical. Generally, the user firstly has to conceive the sequence of steps which conceptually contribute to fulfilling each goal or task. These steps are related to the application considered and do not usually depend on the tools used to work them out. Once the user has decomposed the global goal into basic tasks he has to associate them with accesses to the GIS functions that implement the functionality associated with each basic task. The *Task-to-Functions* refinement process is crucial and requires considerable knowledge of how the GIS structures spatial data and how it is possible to access, integrate and visualise these data. In this work we propose a visual language to support this task-to-function process.

A sophisticated graphical user interface can significantly reduce some problems by creating the bridge between the geo-scientist user's needs and the computer-based instrument. The considerable improvement in hardware technology provides powerful, low-cost, high-resolution graphics workstations. The problem is now how to provide tools to build environments where the users could interact with representations closer to their conceptual world. The use of synthetic dialogues based on gestural input and manipulation of icons can affect the capacity of the user to acquire or infer the conceptual meaning associated with each function or action of the system more simply than by dealing with bytes, files and other computer-related concepts.

Our problem is that GIS users' inquiries often cannot be fulfilled by using a single functionality or command of the system, but involve choosing of a set of co-operating commands, i.e. defining the composition of functions required to achieve the particular task. The specification of the textual procedure which solves the target task may be critical for people who are not used to giving solutions in algorithmical terms, as is the case with most inter-disciplinary GIS users. Therefore, a critical point is the current lack of high-level tools for GIS applications that could help the GIS-user to specify the procedural flow. Indeed, most of the tools generally provided by current commercial GISs are command language interpreters, showing again GIS orientation towards computer or programming-aware users. The "on-the-fly" specification of an algorithm, while interacting with such a complex environment as current GISs, may often discourage the naive non-programmer GIS users. Programming expertise must not be mandatory while using a GIS to solve a non-routine task, and nor is the right solution to collaborate with a "programmer", conceived of as a computer-aware interpreter between the GIS system and the user. In the latter case, the lack of a common language between the "geographer" and the programmer could cause misunderstandings or incorrect use of the system

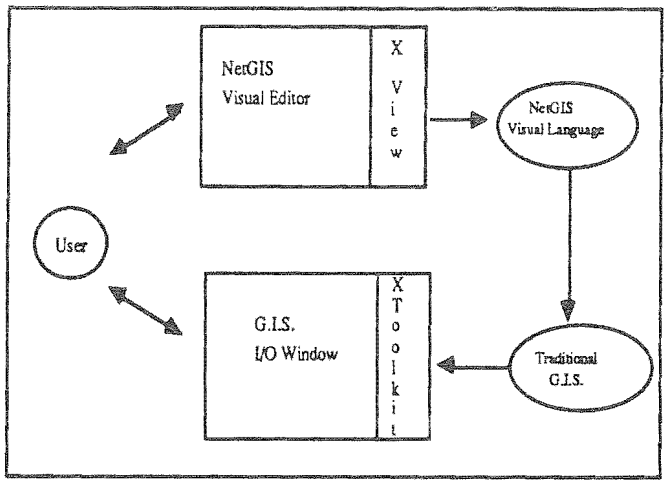

Figure 1. *The NetGIS Environment.*

from a geographical point of view. The application of tools for the design of the procedural flows in the design of user interfaces for GISs should be one of the most important aspects to take into consideration. A similar need for tools arose in software engineering and programming language environments, where visual editing environments have been developed to support the programmer, making it simpler to write syntactically correct and more readable programs[12]. The first step was to define a template-oriented approach to the design of syntax-driven editors. The use of a visual approach leads to such an extreme that the entire program could be codified by visual techniques alone, discarding the textual version in favour of a pictorial or diagrammatic representation. In this approach, tasks could be fulfilled by composing graphical items with each of them associated with application-related functions (in our case GIS functions).

This paper is organised as follows. The components and the GIS functionalities are presented in Section 2. Section 3 describes the visual tool and Section 4 presents the formal specification of the related visual language. Section 5 shows some phases of a work session. Finally, some concluding remarks are reported in Section 6.

2. Components and GIS Functionalities of the Visual Environment

We propose hiding the canonical GIS command language by using a visual language, *NetGIS*. The language is based on building blocks, *modules*, connected in flow diagrams. The module is defined as a basic software element. It implements a medium level functionality which corresponds to one of the typical user basic steps needed in the solution of a task. The module is internally implemented as a procedure of low-level GIS commands. It is not a simple graphical representation of a canonical GIS command. Modules are represented on the screen by icons and the user can interactively build flow diagrams by connecting the output ports of some modules to the input ports of other modules. Using a data-flow paradigm the user builds an application by instantiating modules from the basic set of modules and interconnecting them into a direct acyclic graph, whose nodes are the modules and whose arcs are I/O port interconnections. By interactively composing modules it is possible to solve a complex task in an iterative process composed of a succession of attempts, where the current visual program represented is also the feedback toward the user to remember the steps taken to achieve the result.

A proposal for a task-oriented user interface was also presented by Kirby and Pazner[13]; it is based on a visual approach similar to the NetGIS approach, but the definition of the language and the associated medium level functionalities are only sketched.

NetGIS is designed as a configurable system; it provides a visual environment for the specification, analysis and execution of GIS tasks. The language was built on some basic entities described following an object-oriented approach: the module, the link and the network. A NetGIS program is a *network* defined as a composition of visually represented *modules*; the interconnection between modules is represented by *links*, which are represented visually by arrows. The environment which we are building is described in Figure 1. It is based on a Visual Editor which allows the user to specify an expression of the visual language by direct manipulation techniques. The Visual Editor checks the syntax of the visual program specified by the user; for example, it checks that the output port of a module is connected to an input port which has the same data type. It then translates the

specified visual program in a sequence of calls into a traditional GIS. Final or intermediate results appear on the screen in separate windows managed by the GIS. The user can interact with these windows to manipulate the results with operations such as zooming, panning and so on.

The *module* represents a high level GIS function, implemented as a sequence of GIS commands. We are aware that most of the workstation-based GIS systems have hundreds of functions. It is quite normal to have a large number of GIS functions defined in any GIS implementation. Our idea is to identify the most frequent tasks and to associate them with more abstract functions which can to some extent be customised by providing specific values to the related parameters. The module is characterised by one or more input and output data flows. The module is conceived of as a building block with which the abstraction level of the GIS interface can be increased. Traditional GIS interfaces are closer to the system implementation than to the user's conceptual world. Such low level interfaces allow more flexible specifications but are usually more complex. NetGIS, on the other hand, allows users to specify their requests more closely to their own conceptual model of the GIS.

The composition rules among modules are based on the number and the data type of the input and output ports associated with each module. This approach also provides reusability because if a particular composition of modules provides an interesting processing it can be used by other people without knowing how it works in detail, but just by knowing the I/O and the semantic action associated.

A module is visualised by an interactive graphical representation, the *frame*. The *frame*, a rectangular area, provides a visual representation of the class which the module belongs to by a name and an icon, and contains a set of interactive fields. One or more fields represent the *communication ports*, from which the module receives input data or sends results; each of these I/O ports has a specific data type associated with it. Other interactive fields in the frame are the *mover*, the *info* and the *more*: the first changes the module geometrical position on the screen, the second allows the user to ask for information on the module and its related functionality, the third is used to visualise and modify the parameter values associated with the instantiated module. The transformation of the input data operated by a module may depend on the current setting of a parameter list; the user selects the *more* field to display the module parameters dialogue panel, in order to modify or simply to show the parameter default/current values (an example is shown in Figure 7).

Both iconic and textual information are included in the module frame. Iconic representation is often more intuitive and immediate than a single word to synthesise a concept or a functionality. But, especially when a stable iconic alphabet has not been established, interpreting icons could require considerable intuitive ability on the part of the user to get the correct meaning. The field of GIS is now in such a situation: common iconic symbols have not been sufficiently identified and there are also some GIS operations for which it is difficult to single out an iconic symbolism which could unambiguously recall them. Having both a name and an icon on each module frame is a way to use both of these media, by trying to define an iconic symbolism while using those fundamental terms which hopefully are in the user's common vocabulary, even those who do not speak English.

The following list of available modules has been defined to obtain a (not exhaustive) subset of GIS high-level functions. Defining a basic list is a critical issue, which has not yet been sufficiently taken into account in literature[13]. For each module, a brief description of the semantic action associated, of the number of I/O ports (IP: Input Port, OP: Output Port) and of the type of information transferred is given:

- *Set* interactively selects a coverage in the geo-cartographical data base; it allows interactive browsing in the file system, searching for the area of interest, in a similar way to that provided by common file managers on PCs or workstations. [IP: none; OP: map name].
- *Edit* is used to modify the map received in input and, if needed, to change the name of the modified map. The user can change or delete both graphical and textual features (e.g. nodes can be moved, the line that connects two nodes can be modified, feature nodes can be deleted or substituted); during this operation the topology is generally deconstructed. [IP: map name; OP: map name].
- *Topology* generates the correct topology of the flat data received in input; the input is a map without topological information (created by converting data or by data entry activity), or with corrupted topological information (e.g. because some edit operations have been operated on it). The module creates or rebuilds the topological relations, under the control of the parameters selection operated by the user. [IP: map name; OP: map name].
- *Buffer* generates the buffer areas for the elements in the input map; in other words, it builds an area of influence around the selected feature, returning a new map where the selected linear/point features have been transformed into areal features (e.g. if the input map represents a route, the out map could be the area of noise caused by the route). [IP: map name; OP: polygons map name].
- *Near* implements a spatial analysis functionality (neighbouring): it computes the distance between each

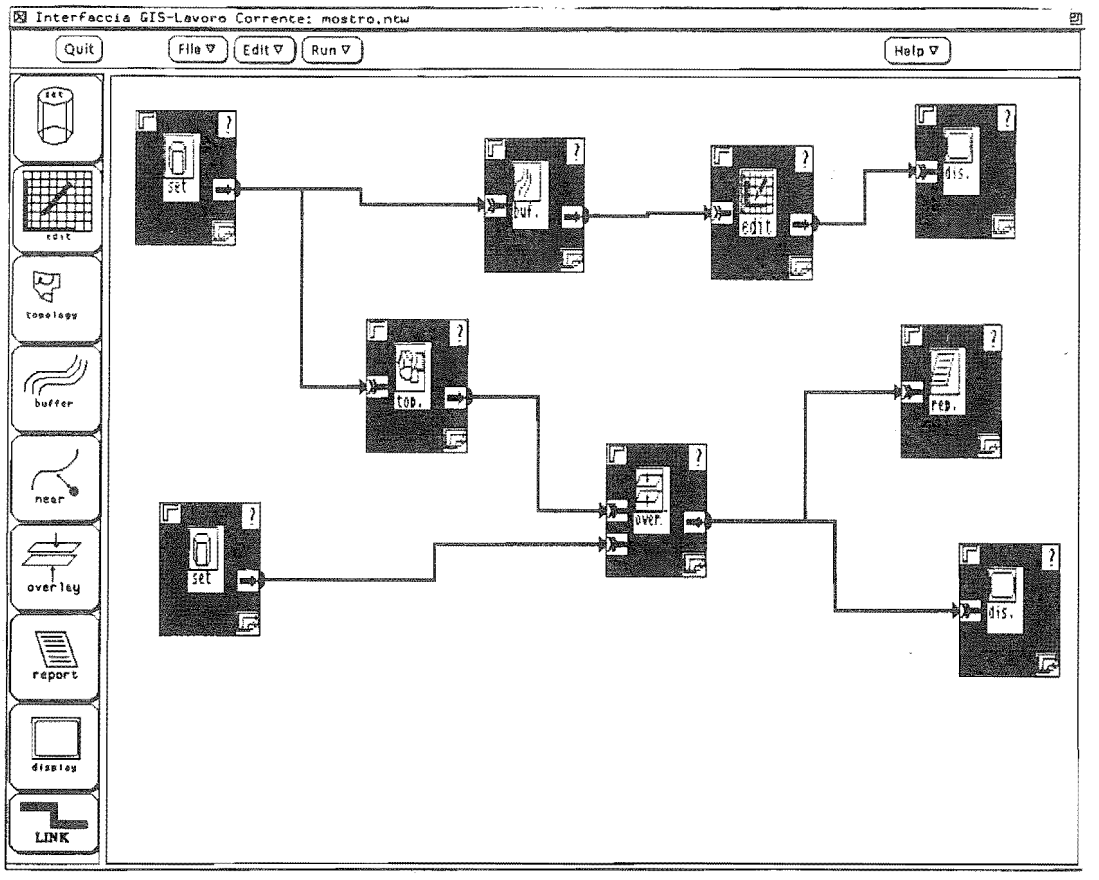

Figure 2. *An example of a network.*

point on the first map and the nearest line or point on the second input map. [IP1: points map name; IP2: points or lines map name; OP: points map name].

- *Overlay* builds the overlapping between maps; it gets two maps in input, with different features or themes related to the same territorial frame (i.e. road network, biomasses, pollution, etc.), and builds the "overlaid" map. [IP1: points map name; IP2: points or lines map name; OP: map name].
- *Display* visualises the carto-thematic values of a map on the screen. [IP: map name; OP: none].
- *Report* builds, and prints on request, a report file on the contents of the map. [IP: map name; OP: none].

If the user needs to perform a function which is not available among those related to the available frames there are three possibilities: the function can be obtained by a specific composition of the available frames; the function is not supported by the underlying GIS so likewise it cannot be supported by the visual environment; the function can be supported by the underlying GIS but needs an extension of the visual language with a new module associated with a new sequence of commands of the underlying textual GIS.

A *link* represents a connection between two modules (Figure 2). It introduces a logic and temporal ordering among modules and determines the visual adjacency. A link is visualised as a directed polyline which connects two I/O ports. Links are interactively created by the user; the user-editor dialogue for the creation of a new link entails firstly selecting the link icon on the vertical menu bar of the Visual Editor (operated, as usual, by positioning the mouse on the icon and then clicking) and then selecting the two I/O ports which the user wants to connect.

Links can be also deleted with the *DeleteLink* entry of the *Edit* pull down menu in the horizontal menu bar of the Visual Editor (described in Section 3). All the delete or edit commands work on the current selected object, therefore the selection of the link to delete has to be made before commanding its deletion. A link can be selected by simply clicking on whichever of the two I/O ports it connects. If a module is moved around, all the links connected to it are redrawn automatically. Two types of

links are supported; *point-to-point*, which indicates a communication between two modules, and *broadcast*, which shares out the output of a module to a set of modules.

A *network* is a valid composition of modules and links which forms a directed acyclic graph. As an example, the network in Figure 2 selects two input maps (*map_1*, *map_2*), using two *Set* modules. The first map is transmitted to a Buffer module and to a Topology module. Then, in the first case, after editing, it is transmitted to a display for visualisation. In the second case an *Overlay* module overlaps *map_1* and *map_2* thus obtaining the final map which is visualised on the display (*Display* module) while the *Report* module stores some results in a different form.

3. The Visual Tool

NetGIS provides a graphical environment to specify visual programs and to control their executions. The system supports:

- visual editing, with direct manipulation specification of NetGIS programs; the editing tool is syntax-driven and it allows the incremental specification of pictures belonging to the language;
- storing and retrieving the specified NetGIS programs on/from secondary storage;
- interpretation of NetGIS programs; a textual representation of the visual expression is derived from the picture: this is checked against static semantic requirements and then executed on the underlying GIS. Users cannot view the code generated by visual programming because our assumption is that they have no background in textual programming so they would have difficulty in understanding it and they could not perform further editing.
- help functionalities, to guide the user and to allow self-explanation of system functionalities.

A NetGIS program consists of a network composed of modules organised in a direct acyclic graph, and therefore only a partial ordering between modules is expressed in the visual program. While the user specifies the network, the editor visualises the current network, manages a data structure that represents the network, and performs syntactic checks such as compatible data types of connected I/O ports and parameter settings. The graphical to textual mapping is performed during network editing, transparently to the user. The resulting textual string can be simply inferred from the above Visual Editor data structures, thus providing a linear string of elements with a total ordering. This ordering can be followed in the execution phase as well. Further syntactical controls are applied on the textual string before the execution to detect other errors, e.g. the missed definition of a link on an I/O module port.

The Visual Editor was implemented with a X toolkit, X View[14]. The visual editor layout (Figure 3) is composed of three areas:

- *horizontal pull-down menu* which specifies the main command groups (File, Edit, Run, Help);
- *vertical menu* where the link and the module classes available to the user for creating new instances are represented by icons;
- *work area* where the user can specify NetGIS programs by direct manipulation.

The network specification proceeds by subsequent instantiations of modules and links. Selecting one of the icons in the vertical menu creates a new instance of a module and visualises it on the work area, at a position chosen by the user. The instantiation of a new link works in the same way: the user firstly selects the link icon on the vertical menu and then selects the two I/O ports that have to be connected by the link. By instancing links the user defines logical and temporal orderings between modules. During this phase syntactical controls are applied to avoid specifying inconsistent connections. If there is an incorrect specification (i.e. two links entering in the same input port, or a link connecting two ports with a different data type) an error message is generated by the editor and the instantiation is rejected.

The current network could be modified by deleting and/or adding modules, by altering the position of the modules, or by redirecting some links, thereby changing the logical ordering between modules. By selecting interactive fields of each instantiated module, the system can visualise information on the application semantic or the current values of the module parameters, whose values can also be changed. The specification of the current NetGIS program can be saved for further use at any moment.

The visual environment was designed following the object-oriented paradigm, and thus provides good abstraction mechanisms and extendibility by using inheritance, data hiding and reusability. Links, modules and networks are the basic object classes which have been identified for the performance of the visual environment. A module is an object with two categories of attributes: *status* and *appearance*. The *status attributes* are: the class; the semantic task associated; a set of parameters, to modify or customise the actual semantic function associated with each instantiated module; the execution state (ready, executed, running or data waiting); links-connected, a boolean to indicate if associated connections have been defined; the links list, the list of the defined connection; and the list of the I/O ports. The *appearance*

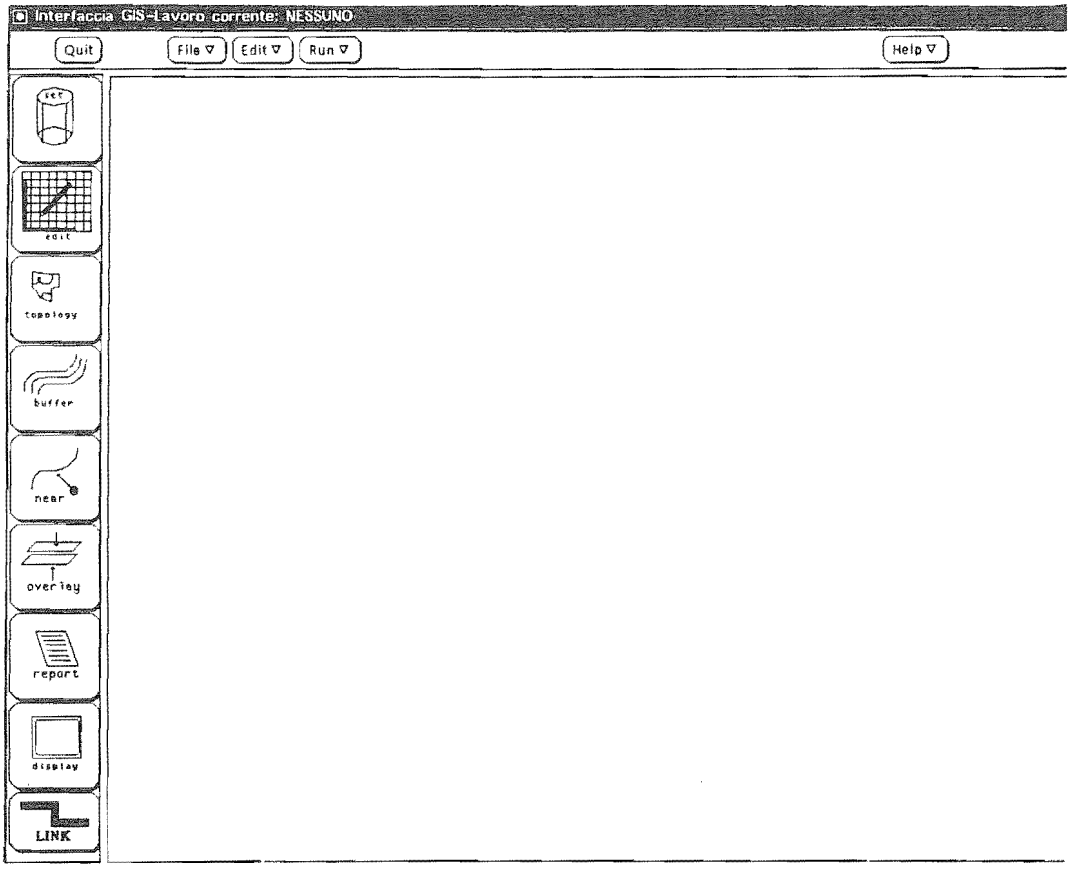

Figure 3. *Layout of the NetGIS editor.*

attributes include the dynamic information for drawing the graphical items associated with the module. The methods associated with the general module class are: creation, deletion, visualisation, attributes update, selection and execution.

A visual NetGIS program can be executed under two different policies: *user-driven* and *dataflow*. The user selects one of the two possible ways by indicating the associated option in the *Run* command of the horizontal main menu.

In the *user-driven* policy, the system gives users full control over the next module to be executed, by asking for a step-by-step explicit selection of the next module. If the chosen module is in an executable state (i.e. if it has the data input ready on the input ports) the request is satisfied; otherwise, the system communicates that the choice was wrong and asks for a new selection to continue or to stop the run. Moreover, the user can dynamically interact with the network at execution time too, by modifying the execution ordering of the modules depending on the partial results, or by stopping the execution of the network if the resulting data are not useful.

This policy is particularly useful during the specification of a new application; when the user defines a new network and is not sure of the effectiveness of the solution, the evaluation of the intermediate results can be particularly useful.

Under the *dataflow* policy the system executes a network by selecting one of the total orderings between modules allowed by the topology of the specified network: execution is thus completely automatic. Under this second policy a faster, unattended execution is allowed.

The prototype uses ESRI's *ArcInfo* as its underlying GIS[15]; an AML (the ArcInfo command language) subroutine was thus defined for each module to implement the task associated with the module. Whichever policy is chosen, each module to be executed is "translated" into the corresponding set of GIS commands, taking into account the non-default parameter values set by the user during the specification of the visual program; this set of commands is transferred to the underlying active GIS which will execute it.

Most of the current visual environments, especially in computer graphics or image processing, are based on an interpretation of the dataflow paradigm where data transmitted between modules are the whole data managed by the application. For example, visualisation applications such as aPE[7] or AVS[6] have modules for the selection or generation of a graphical model and a number of different visualisation modules. When the user connects a couple of the previous modules, the graphical model representation is transmitted from the generation module to the visualisation one. This data transmission could become a bottleneck when complex datasets have to be visualised or managed. The approach chosen in the design of NetGIS is different: given a link connecting a module couple *msend mrec* the system only transmits from *msend* to *mrec* reference information which will be sufficient to univocally identify the dataset which *mrec* will read in input. This approach makes it possible to reduce the amount of data transmitted, and this is especially valuable in the field of GIS where there is generally a vast amount of data to manage.

4. The Formal Specification of the Visual Language

Graphical or visual editors based on direct manipulation provide convenient environments for the specification of visual programs but can easily give rise to mistakes. A formal specification of the visual language is required to prevent implementation errors during the design of the visual editor. The purpose is to precisely identify which graphical representations are associated with a semantic effect in this application. The specification should be precise, compact and readable. But while the definition of the syntax and semantics of textual languages have been widely studied and stable techniques are ready to use, a similar agreement has not yet been found in the field of visual languages. Two approaches have been investigated for the formal definition of NetGIS.

The syntax-driven Visual Editor was designed following the *indirect approach*, where the language is defined using traditional textual grammars. Expressions of the textual syntax are mapped to expressions of the graphical syntax by a function that incorporates the graphical relationships among the iconic elements of the language. In order to make the graphical relationships among the visual language constructs more evident a *direct approach* was also applied, based on the picture layout grammars[16], a specific grammar type which can directly define the graphical syntax of a visual language.

In the *indirect approach*, the grammar for textual NetGIS is defined as a common BNF grammar. The grammar is defined as follows with the usual 4-tuple [*N*, *T*, *S*, *P*] with *N* the set of non-terminal symbols, *T* the set of terminal symbols, *S* the initial symbol and *P* the set of derivation rules. The non terminal symbol *Mij* represents modules with *i* input ports and *j* output ports; a terminal *lo* is associated with each link connected to an output port, and analogously *li* is associated with a link connected to an input port; the sequence *lo*)*li* therefore represents a link that joins two modules. The terminals *Set*, *Edit*, *Topology* and so on, represent specific NetGIS modules.

N = {IN, IN', M01, M10, M11, M21, OUT, OUT', PROG, PROG'}
T = { Set, Edit, Topology, Buffer, Near, Overlay, Display, Report, li, lo, (,), [,], ||}
S = PROG
P = { 0: PROG → (IN) OUT;
1: IN → (IN) li M11 lo;
2: IN → (IN), (IN) li M21 lo;
3: IN → M01 lo;
4: OUT → [(PROG') || OUT'];
5: OUT → li M10;
6: PROG' → (IN') OUT;
7: OUT' → (PROG') || OUT';
8: OUT' → (PROG');
9: IN' → (IN') li M11 lo;
10: IN' → (IN'), (IN) li M21 lo;
11: IN' → lo;
12: M01 → Set;
13: M11 → Edit | Topology | Buffer;
14: M21 → li Overlay | li Near;
15: M10 → Display | Report;}

The syntax analysis process consists in firstly mapping a visual program into textual format. The textual string is built by replacing each module with the associated terminal (i.e. Set, Edit, Topology, etc.); the network is linearized by using an algorithm that visits the graph in left-to-right, top-down order. If we consider a network which allows us to select two input maps using two Set modules, the first map is transmitted to a Buffer module and then an Overlay module overlaps its result and the second map, thus obtaining the final map which is visualised on a Display, while a Report module stores some results in a different form. The textual string obtained by the mapping function applied on this network is:

S = (((Set lo) li Buffer lo), (Set lo) li li Overlay lo) [((lo) li Report) || ((lo) li Display)]

Finally the program in textual NetGIS is translated into a sequence of calls to the underlying GIS command language.

A *direct approach* was also used to define the NetGIS syntax. It is based on picture layout grammars (PLG), a particular type of grammar introduced by Golin and Reiss[17] and based on the attributed multiset grammars (AMG). A multiset is a set where elements can be repeated and the AMG is similar to a context-free grammar but the right-hand sides of the productions are considered to be unordered collections of symbols, rather than strings. Each AMG production is a triple (*R*, *S*, *C*),

where R is a rewrite rule, S is a semantic function which computes the attribute values of the left-hand side from the attribute values of the right-hand side, and C is a constraint defined over the right-hand side attributes which indicates when the production can be applied.

PLG are defined as AMG where a production corresponds to a *picture composition operator* and the attributes represent spatial information on a picture element (terminal or aggregate). A sample PLG production is therefore:

A → Op(B, C); (rewrite rule)

A.attr = FuncOp (B.attr, C.attr) (semantic function)

where: PredOp(B.attr, C.attr) (constraint)

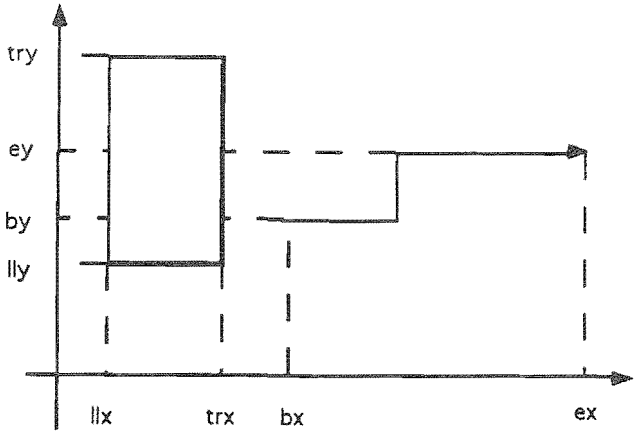

Figure 4. *The presentation attributes of module frames and links.*

Terminal symbols of the NetGIS PLG are the basic graphical elements of NetGIS (links and modules); attributes are defined on each element class, to describe its pictorial and spatial aspect, and the productions use spatial operators to describe element composition. Each NetGIS PLG symbol (both terminal and nonterminal) has a set of spatial attributes (Fig. 4). *Module attributes* are the position of the two corners of the rectangular area of the frame, lower-left (*ll*) and top-right (*tr*), and the position and type of each I/O port. The graphical representation of a link depends on the position of its beginning point *b* and end point *e* and it is drawn by tracing the three segments [(bx, by) ((bx+ex)/2, by)], [((bx+ex)/2, by) ((bx+ex)/2, ey)] [((bx+ex)/2, ey) (ex, ey)], the last of them terminating with an arrow. The *link attributes* are therefore the former *b* and *e* points. The *network attributes* are analogous to the module ones, i.e. the two corners *ll* and *tr* of the network bounding rectangle.

The operators working on NetGIS symbols are:

- *TouchesLeft*(*B*,*C*), where B is a link, C is a network and

  (C.trx = B.Bx) ∧ (C.lly < B.by < C.try);

- *PointsTo*(*B*,*C*), where B is a link and C is a network and

  (C.llx = B.ex) ∧ (C.lly < B.ey < C.try);

- *AdjacentTo*(*B*,*C*), where B,C are networks and (B.trx = C.llx);

- *Contains*(*B*,*C*)), B contains C, where B,C are networks and

(B.llx < C.llx) ∧ (C.trx < B.trx) ∧ (B.lly < C.lly) ∧ (C.try < B.try).

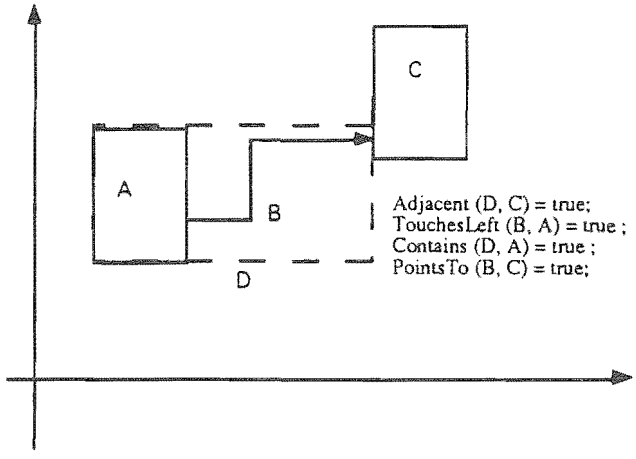

Figure 5. *Evaluation of operators.*

In Figure 5 these operators are evaluated on some terminal and composite symbols. The complete set of productions for the specification of NetGIS syntax via PLG is defined in [18]. This approach is tool-supported[17]: it is possible to provide a PLG, then the spatial parser uses the grammar definition of the visual language to parse the graphical representation generated by the user and recover the underlying structure of the program.

Comparing the two approaches to a visual syntax description is not easy. By definition, picture layout grammars facilitate a more formal definition of visual language syntax as a whole, without requiring mapping functions which generally embody most of the visual cues of the language. On the other hand, the syntax-driven editor, common in the indirect approach, makes the specification of syntactically correct programs easy. Using a direct approach, the programmer can inde-

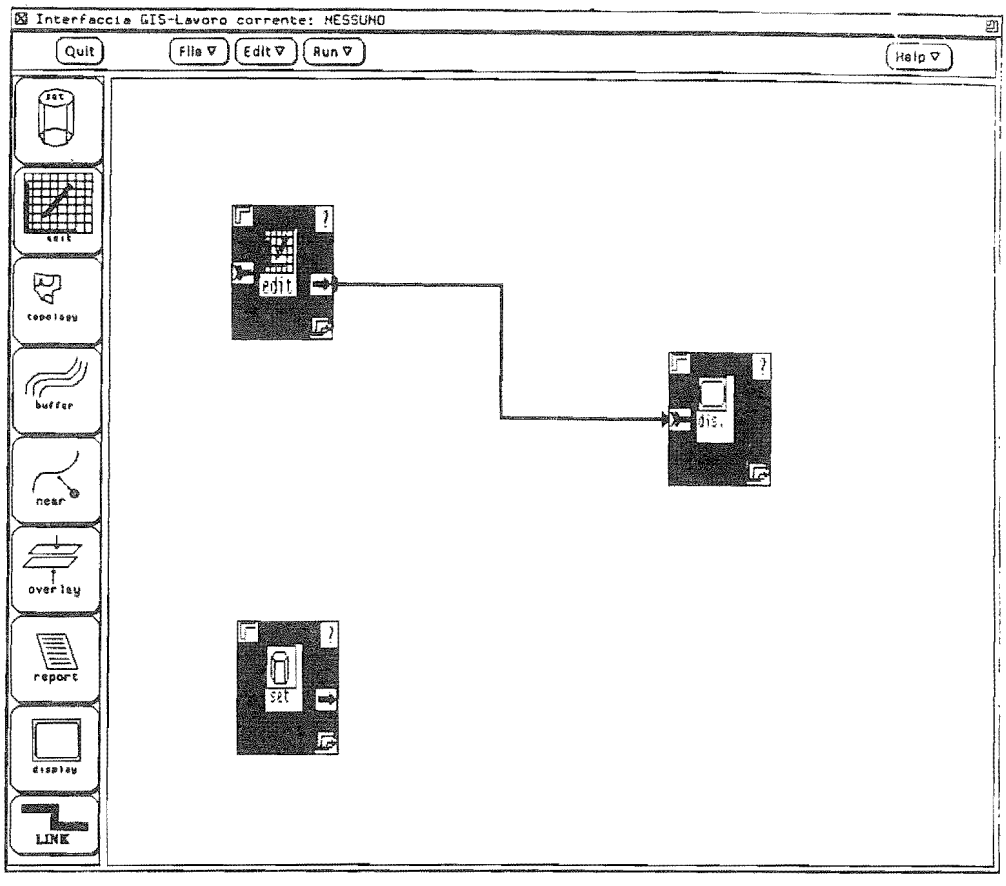

Figure 6. *Erroneous specification of a link.*

pendently specify the graphical program, without the constraints that the syntax-driven editors sometimes impose; the visual program is then passed to a spatial parser that builds and checks the associated syntactic structure.

One aspect that is not covered by either of the approaches is how to cope with the possible dynamic aspect of a visual language (VL). *Interactive VLs* can be defined as languages which allow interactive constructs, i.e. programs whose syntactic content cannot be completely described by a static visual representation of the program itself. NetGIS resembles such an interactive VL if we look at the module parameter setting field and see how it works for. Although the actual parameter values are not directly represented, their values affect the syntactical correctness of the network and its associated semantics. The user, using the visual editor, can assign values to the parameters interactively, by selecting (pointing and clicking) the *more* field of a module frame. A dialog window is then visualised, which contains the list of the module parameters and their current values; the user can modify them by typing new values. Interactively hiding/showing part of a specification could be seen as an important potentiality of VL (or, at least, of the supporting VL environments) but the consolidated syntax description techniques, such as those presented here, cannot describe more than the static part of the visual language. In[19] there is a discussion about the possibility of using LOTOS in order to overcome these problems. The LOTOS formal notation combines process and data algebra: the first can be used to describe the dynamic behaviour of the graphical elements, the second to define their internal state and the related appearance on the screen.

5. A Work Session Example

Figure 3 shows the layout of the system at startup time. Figure 6 shows a work session where the user firstly instantiates two modules, an *edit* and a *display* module, and a link which connects the output port of the edit to the input port of the display. Then the user instantiates a *set* module and if he tries to connect the output port of this module to the input port of the display, the Visual Editor detects the erroneous request of the user, communicates the error to the user and rejects the request.

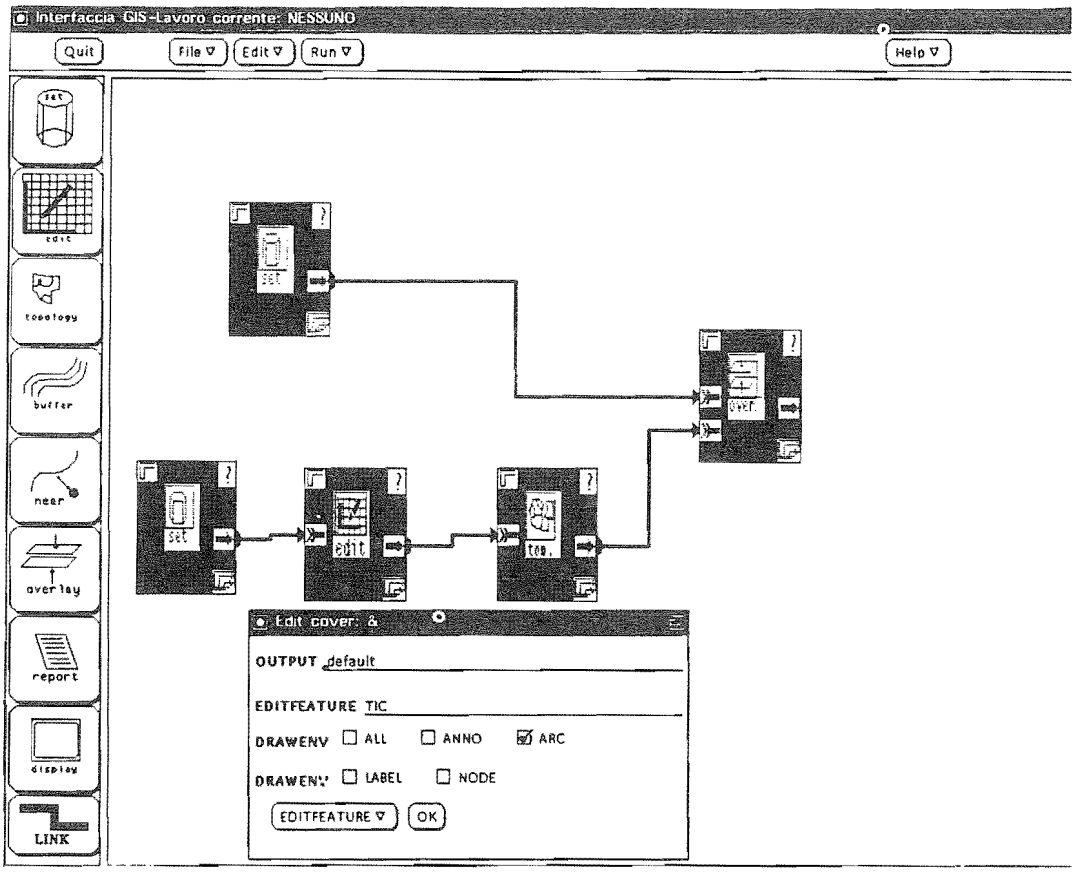

Figure 7. *The parameter setting dialog window.*

The erroneous link is therefore not drawn by the Editor. In the example if Figure 7 the programmer has instantiated five modules and has correctly connected some of them. From an application point of view the network in Figure 7 may be conceived of as follows. The two *set* modules select two distinct input maps (map1, map2), with different geo-territorial features. Map1 could represent both some point land features such as the pollution sources due to some waste sites and an areal feature associated with each waste site, which will represent the polluted area. Map2, a line cover, could represent some linear land features such as the river network present in the same region. Map2 is then given in input to the *Edit* module, which allows the user to modify the river network geometry (e.g. by changing the position of an incorrect node-junction between a river and one of its affluents, or the course or shape of a canal). After the editing session, the modified coverage needs to be submitted to the *Topology* module in order to reconstruct the topological consistency (i.e. the relations between the geometrical entities of the map). The *Overlay* module creates a new map by superimposing the two input maps, map1 and the edited map2 (which must have a compatible map scale). The resulting map produced by the *Overlay* module has a consistent topology and shows a region inside which the hydrographic network is spatially related to the waste site disposals, and where the river sections flowing into polluted areas could be simply identified.

In the example in Figure 7 the user's selection of one *more* field of the Edit module caused the visualisation of the corresponding parameters dialog window. The user can now interactively modify the current parameter values shown; the module internal status will be consistently updated.

6. Conclusions

This paper discusses the design and specification of a visual language using *NetGIS* as an example. This is a visual language and an associated graphical programming environment proposed to reduce the complexity of the generical GIS application interface. The visual language improves the management of the complexity of the *task-to-function* process, i.e. how to fulfil a specified user task by using the low-level functionalities provided

with the GIS. The proposed language, based on the *module* concept, is an attempt to deal with this problem by defining a higher abstraction level GIS functionality set. The language is defined as a set of interacting objects (modules and links) whose interconnection specifies a procedural flow graph. The result is a visual programming environment, easy to use and closer to the conceptual model of the user than the usual graphical GIS interfaces.

A formal specification of the visual language was developed in order to precisely indicate which graphical representations can be associated with a call to a sequence of GIS functionalities. The power of the NetGIS approach clearly depends on the correct definition of the basic modules set. A potential set is defined in this paper and was implemented in the prototype. In order to increase flexibility, a possible extension of the system could be to develop a tool for the extension of the set of basic modules, thus allowing users to define their own custom set of modules. The definition of new module classes and their integration in the programming environment should be possible without knowledge of the software implementation of the system.

Acknowledgements

We are grateful to Ernesto Bronzini, Carlo Magnarapa and Sandro Mazzotta for valuable discussions and for their support in the system design.

References

1. S.K. Chang, "Visual Languages: a Tutorial and Survey", *IEEE Software*, January, pp. 29–39 (1987).
2. N.C. Shu, "Visual Programming", Van Nostrand Reinhold Books, New York (1988).
3. B.A. Myers, "Taxonomies of Visual Programming and Program Visualization", *J. of Visual Languages & Computing*, Academic Press, **1**(1), pp. 97–123 (1990).
4. E.P. Glinert, "Visual Programming Environments", IEEE Computer Society Press (1990).
5. P.A. Haeberli, "ConMan: A Visual Programming Language for Interactive Graphics", *ACM Computer Graphics*, **22**(4), Aug. pp. 103–111 (1988).
6. C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization", *IEEE Computer Graphics & Applications*, July, pp. 30–42 (1989).
7. D. Dyer, "A Dataflow toolkit for the Visualization", *IEEE Computer Graphics & Application*, pp. 60–68 (1990).
8. M.K. Leong, S. Sam, D. Narasimhalu, "Towards a Visual Language for an Object-Oriented Multimedia Database System", in *Visual Database Systems* (Proceedings of the IFIP Conf.), ed. T.L. Kunii, Elsevier Publisher, pp. 465–495 (1989).
9. D.D. Hils, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages", *Journal of Visual Languages and Computing*, **3**, pp. 69–101 (1992).
10. C. Crimi, A. Guercio, G. Nota, G. Pacini, G. Tortora, M. Tucci, "Relation Grammars and their Application to Multi-dimensional Languages", *Journal of Visual Languages and Computing*, **2**, pp. 333–346 (1992).
11. J.M. Carrol, J.R. Olson "Mental Models in Human-Computer Interaction", in *Handbook of Human Computer Interaction*, M. Helander (ed), North Holland, pp. 45–65 (1988).
12. A.L. Ambler, M.M. Burnett, "Influence of Visual Technology on the Evolution of Language Environments", *IEEE Computer*, October, pp. 9–22 (1989).
13. C.K. Kirby, M. Pazner, "Graphic Map Algebra", *Proc. 4th International Symposium on Spatial Data Handling*, Zurich, 413–421 (1990).
14. *XView Programming Manual*, O'Reilly Associates, Inc., 566 (1989).
15. *User Manual – ArcInfo*, ESRI, Redlands CA (1986).
16. E.J. Golin, "Parsing Visual Languages with Picture Layout Grammars", *Journal of Visual Languages and Computing*, Academic Press, **2**, pp. 371–393 (1991).
17. E.J. Golin, S.P. Reiss, "The Specification of Visual Language Syntax", *Journal of Visual Languages and Computing*, Academic Press, **1**(2), June, pp. 141–157 (1990).
18. E. Bronzini, S. Mazzotta, "Tecniche di Programmazione Visiva applicate ai Geographical Information Systems", Tesi di Laurea, Dipartimento di Informatica, Universita' degli Studi di Pisa, December (1991).
19. F. Paterno', "A Formal Specification of Appearance and Behaviour of Visual Environments", *Software Engineering Journal*, May pp. 154–164 (1993).