Journal of Visual Languages and Computing (2002) **13,** 573–600
doi:10.1006/S1045-926X(02)00025-3 available online at http://www.idealibrary.com on IDEAL®

A Classification Framework to Support the Design of Visual Languages

G. Costagliola*, A. Delucia†, S. Orefice‡ and G. Polese*

**Dipartimento di Matematica ed Informatica, Università di Salerno, 84081 Baronissi, Salerno, Italy, E-mail: {gencos, giupol}@unisa.it, † Dipartimento di Ingegneria, Università del Sannio, 82100 Benevento, Italy, E-mail: delucia@unisannio.it and ‡ Dipartimento di Informatica, Università di L'Aquila, 67100 L'Aquila, Italy, E-mail: orefice@univaq.it*

Received 30 January 2001; accepted 18 March 2002

An important step in the design of visual languages is the specification of the graphical objects and the composition rules for constructing feasible visual sentences. The presence of different typologies of visual languages, each with specific graphical and structural characteristics, yields the need to have models and tools that unify the design steps for different types of visual languages. To this aim, in this paper we present a formal framework of visual language classes. Each class characterizes a family of visual languages based upon the nature of their graphical objects and composition rules. The framework has been embedded in the Visual Language Compiler–Compiler (VLCC), a graphical system for the automatic generation of visual programming environments.

1. Introduction

The widespread use of visual systems has brought the need to have tools supporting the definition and implementation of visual language environments [1–9], each based on a particular visual language specification method. These specification methods come in several forms, which makes it difficult to systematically compare them and to abstractly classify visual language. There has been an attempt to classify existing approaches based on their expressive power. In particular, several formalisms have been compared and classified according to their formal properties in order to provide a comprehensive Chomsky-like hierarchy of visual languages [10].

Currently, there are three main approaches to visual language specification: the grammatical approach, the logical approach, and the algebraic approach. The grammatical approach is based on grammatical formalisms extending the traditional rewriting mechanism used in string language specification, and uses geometric relationships between the objects to be rewritten. The logical approach uses first-order mathematical logic or other forms of logic from artificial intelligence. Logical techniques are usually based on spatial logics, which axiomatize the possible relationships between objects. Finally, the algebraic approach uses algebraic specifications consisting of composition functions

constructing complex pictures from simpler picture elements. In this paper we will mainly rely on the grammatical approach.

In the last decades several techniques for modeling visual languages syntax have been created, and these have been thoroughly analyzed and compared [11]. It turns out that two main methods can be used to represent visual language sentences: *relation-* and *attribute-based*. The former describes a sentence as a set of graphical objects and a set of relations on them. The attribute-based representation instead conceives a sentence as a set of attributed graphical objects.

In this paper we present a framework of *visual language classes*. Each class characterizes a family of visual languages in terms of the syntactic attributes of their graphical objects and the spatial relations that can be used to compose visual sentences. As an example, a graphical object in the family of flow graph languages is a box with syntactic attributes identifying specific points on the box edge; typical relations are interconnections, which are used to link graphical objects through their attaching points and are visualized by polylines. On the other hand, a graphical object in the family of icon visual languages is an icon whose syntactic attributes are the coordinates of its centroid, and the relations are spatial compositions (such as horizontal and vertical concatenation). These two examples reflect the two generally used modalities to compose visual sentences: by connecting graphical objects, or by spatially arranging them. As a consequence, the classes of the framework are distinguished in two categories: *connection-based* and *geometric-based*.

It is worth noting that most visual languages may be modeled according to different classes of the framework. The appropriateness of the choice depends on how naturally the chosen class describes the language and on specific needs of the visual language designer. In the paper we analyze a wide range of existing visual languages and characterize them according to the class that in our opinion is more appropriate. Also, there are specific visual languages exhibiting syntactic features of different classes, such as box inclusion, links between boxes, horizontal and vertical concatenations. Statecharts are a valid example of this type of visual languages [12]. For this reason, we also introduce the concept of *hybrid modeling* to integrate features from different classes.

The notion of visual language class is embedded in the Visual Language Compiler–Compiler (VLCC) that inherits concepts and techniques of traditional compiler generation tools, like for instance YACC [13], and extends them to the visual field [3, 4]. VLCC provides a visual language designer with visual assistance tools to help him/her during the definition of the graphical objects, the syntax and the semantics of the language being designed. The visual language definition tools are parametric with respect to the visual language class. The designer is able to select a suitable class for the visual language being designed and use the visual grammar editor for that class. For hybrid visual languages the designer can select more than one class. The result of the visual language definition process consists of an integrated visual environment comprising a graphical editor, customized from the class-specific editor template, and a compiler for the designed visual language.

The paper is organized as follows. Section 2 introduces preliminary concepts underlying the proposed framework. Section 3 presents the classification framework, whereas in Section 4 we provide a wide characterization of existing visual languages, classified according to the proposed framework. Section 5 describes the VLCC system, and the way the framework is implemented within it. Finally, concluding remarks are outlined in Section 6.

2. Visual Language Syntax

In this section we provide an overview on basic notions of visual language syntax used in our framework according to the attribute-based approach [11]. A detailed discussion on this topic can be found in [4].

A visual language may be conceived as a collection of visual sentences given by *graphical objects* arranged in the two- or higher-dimensional space. Syntax of visual languages is described through the graphical objects of the language (the vocabulary), the relations used to compose the sentences, and a set of rules defining the visual sentences belonging to the language. The graphical objects of a visual language vocabulary are characterized by a set of attributes that can be classified as *graphical attributes, syntactic attributes*, and *semantic attributes*.

Graphical attributes characterize the appearance of the object. Typical graphical attributes are position, size, shape, color, name, etc. The role of these attributes is to convey information for the spatial arrangement of the object on the screen. In general, they do not convey information about the syntactic correctness of the composed visual sentence. Syntactic attributes are used to relate graphical objects in order to form visual sentences. A set of graphical objects forms a visual sentence when all of their syntactic attributes have been instantiated. Syntactic analysis techniques can be used to check whether the syntactic attributes have been instantiated according to a proper set of relations, i.e. to check whether a visual sentence belongs to a given visual language. It is worth noting that a syntactic attribute can also be a graphical attribute. For example, the position of an icon is both a graphical and a syntactic attribute. Semantic attributes are used to associate semantics to a graphical object; they can be used either to provide a logic interpretation of a visual sentence or to translate it into a target language by using syntax-directed translation schemes.

The types of syntactic attributes and the types of feasible relations that can be applied to them to compose visual sentences are strongly related and characterize a *visual language class*. As an example, let us consider the visual sentence in Figure 1 representing a flowchart. It can be modeled as the interconnection of the graphical objects *start*, *predicate*, *function*, and *halt*.

Each object has a pre-defined set of attaching points as syntactic attributes. The attaching points are connected through polylines that visually depict the control-flow relation among objects. The semantics associated to the attaching points of these graphical objects defines the direction of the connections.

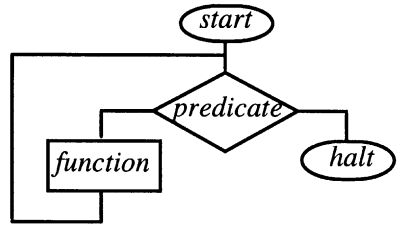

Figure 1. A flowchart

If 'a', 'b', and 'c' indicate the three interconnections (*start, predicate, function*), (*predicate, function*) and (*predicate, halt*), respectively, then the syntactic attributes of the graphical objects in the sentence of Figure 1 are instantiated as in Table 1.

A class of visual languages defines a family of visual languages of the same type. As we will see, we can provide a common design framework for visual languages of the same class. Thus, when designing a visual language it can be useful to first analyze its characteristics in order to associate it to an appropriate class. For example, the visual sentence in Figure 2 shows the layout of the front page of a two-column paper. This sentence belongs to a visual language whose graphical objects are rectangular boxes. In this case, the coordinates of the upper-left and lower-right corners of the rectangle can be used as syntactic attributes. These convey information on the size and the position of a graphical object. Moreover, visual sentences can be composed by using geometric relations, such as spatial inclusion, horizontal and vertical arrangement.

The two examples shown in Figure 1 and 2, respectively, highlight the two basic modalities to compose visual sentences, i.e. by explicitly connecting or by spatially arranging graphical objects. In the first case, the syntactic attributes of graphical objects are instantiated by means of link relations (the connections between them). In the second case, the syntactic attributes are implicitly instantiated when a graphical object is placed in the Cartesian plane, whereas the relations between two objects are derived from their relative positions.

Table 1. An attribute-based representation of the flowchart in Figure 1

Object name	*attaching point 1*	*attaching point 2*	*attaching point 3*
start	a	—	—
predicate	a	b	c
function	b	a	—
halt	c	—	—

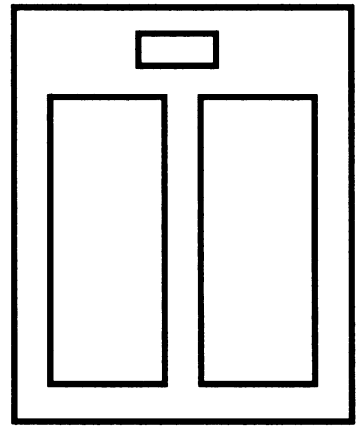

Figure 2. A sample box sentence

It is worth noting that the same visual language may be associated to different classes. Hence, the language can be modeled according to different design frameworks. For example, the graph shown in Figure 3 can be modeled either by explicitly interconnecting nodes and arrows (both seen as graphical objects) or by arranging them in the space. In the last case, the position of an end point of an arrow coincides with the coordinates of a point on the circumference of a node. Here the implict relations among nodes and arrows are geometrically derived.

The choice of an appropriate class to model a visual language may depend on its intrinsic characteristics. For example, a graph visual language is more naturally modeled through a class defined upon the connection-based visual sentence representation method rather than the one defined upon the geometric-based method.

3. Classes of Visual Languages

In this section, we provide a framework of visual language classes. Classes can be divided into two categories: *connection-* and *geometric-based*. These reflect the two basic modalities used to compose visual sentences, i.e. by connecting or spatially arranging graphical objects. In the following we characterize some basic classes belonging to these two categories. Each class of visual languages is completely specified by the type of syntactic attributes associated to the graphical objects and the types of relations that can be used on them to compose visual sentences. This classification allows us to characterize a wide range of existing visual languages, as shown in Section 4.

3.1. Connection-based Classes

The visual sentences described by connection-based classes are formed by a set of interconnected graphical objects. The syntactic attributes of a graphical object can be single attaching points or sets of attaching points. In the latter case, the sets can be discrete or continuous (such as lines, areas), and we will refer to them as *attaching regions*. The value of an attaching point or region participating to a connection is given by a unique identifier for that connection (see Table 1). Table 2 summarizes the characteristics of some basic connection-based classes that we will describe in this paper. It is worth noting that an attaching point is a special case of an attaching region containing only one point.

Connections may be visualized in different ways. In this paper, we focus on two visualization modalities: *overlapping* and *linking*. In the former, connections are visualized by

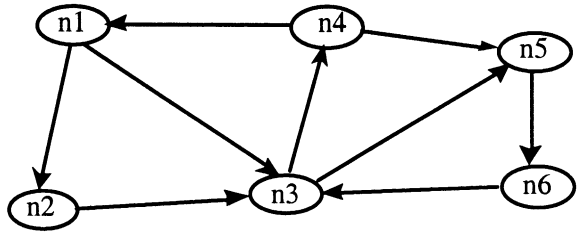

Figure 3. A sample graph

Table 2. Basic connection-based visual language classes

Class	Realtions	Syntactic attributes
Plex	Any connection	A pre-defined number of attaching points
Graph	Any connection	A pre-defined number of attaching regions

overlapping attaching points/regions, whereas in the latter by explicit links between them. Figure 4 shows two examples of overlapping [Figure 4(a) and (b)] and two examples of link [Figure 4(c) and (d)]. Figure 4(a) shows two lines forming the V letter by overlapping their attaching points, while the graphical objects in the puzzle of Figure 4(b) are connected by plugging protrusions ('knobs') into correspondingly shaped indentations ('sockets'). Figure 4(c) shows two graphical objects of a flowchart whose attaching points are connected through a link; similarly, the attaching regions (the circumferences) of the two nodes in Figure 4 (d) are linked to form a simple graph.

The selection of a connection type depends on the characteristics of the visual language under study. As an example, edge-labeled graphs and directed graphs may be modeled as sets of nodes and edges where the circumference of a node (attaching region) can be overlapped to one of the two end points of an edge (attaching point). The choice of this representation is due to the fact that edges carry their own information, such as labels and direction, and therefore they need to be represented through graphical objects. On the other hand, unlabeled undirected graphs could be directly modeled as sets of nodes joined through polylines ('links') representing relationships between nodes.

3.1.1. Class Plex

The class *plex* is suitable for modeling graph-structured visual languages, with the limitation that each terminal graphical object can only have a fixed number of connections.

Syntactic attributes

Each graphical object has a pre-defined set of points on its image as syntactic attributes; these points are named *attaching points*.

According to the terminology introduced in [14], we will refer to graphical objects of the class plex as *NAPEs* (N Attaching Point Entities) and use the integers in the subrange (1, *n*) to identify the *n* attaching points of a NAPE.

Relations

The relations are *plex interconnections* denoted by *JOINT* and defined as follows.

Definition 1. Given the NAPEs *a* and *b*, then *a* JOINT(h, k) *b* holds if and only if the attaching point h of *a* is connected to the attaching point k of *b*.

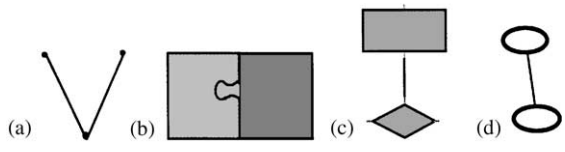

Figure 4. Two types of connections: *overlapping* (a), (b) and *link* (c), (d)

As an example, Figure 5(a) shows the graphical objects of a logical circuits language, whereas a simple logical circuit composed by joining the attaching points of the NAPEs is depicted in Figure 5(b). The JOINT relation is visualized through links.

Table 3 instantiates the attaching point of NAPEs in the visual sentence with the link values.

3.1.2. Class Graph

The class *graph* is suitable for modeling general graph-structured visual languages.

Syntactic attributes

Each graphical object has a pre-defined set of regions on its image as syntactic attributes; these attributes are named *attaching regions*.

Relations

The relations are *graph interconnections*, denoted by *JOINT* and defined as follows.

Definition 2. Given the graphical objects *a* and *b*, then a *JOINT*(h, k) *b* holds if and only if the attaching region h of *a* is connected to the attaching region k of *b*.

Figure 6 shows a simple lattice. The only graphical object is the node that has two attaching regions, the upper semi-circumference (light line) and the lower semi-circumference (thick line).

The JOINT relation is visualized through links joining the lower semi-circumference and the upper semi-circumferences of two nodes, respectively.

3.2. Geometric-based Classes

In geometric-based classes of visual languages a visual sentence is described as a set of graphical objects spatially arranged in the Cartesian plane. Several spatial composition rules can be used to form visual sentences. Examples include metrics-based relations,

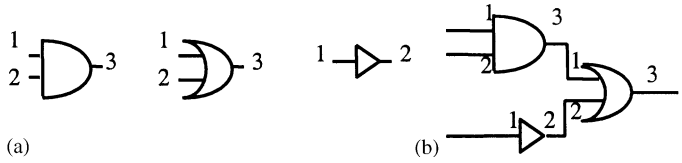

Figure 5. NAPEs of a logical circuits language (a) and a simple visual senetence (b)

Table 3. Instantiation of the attaching point of NAPEs in Figure 5(b)

Object name	*attaching point 1*	*attaching point 2*	*attaching point 3*
and	—	—	a
or	a	b	—
not	—	b	—

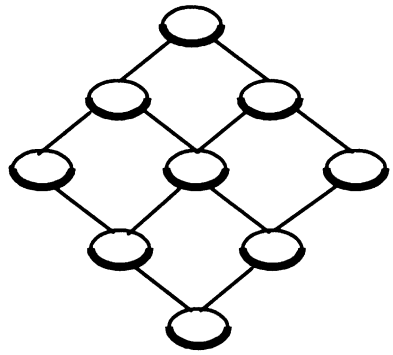

Figure 6. A simple lattice

such as horizontal and vertical concatenation[a], spatial inclusion, adjacency and intersection. The syntactic attributes of a graphical object are the coordinates of a set of points suitable to relate the objects according to a spatial composition rules above. Syntactic attributes are automatically instantiated when placing the graphical object in the Cartesian plane.

As an example, *Vénn diagrams*, used to depict set relations, such as inclusion and intersection, can be modeled according to geometric-based classes (see Figure 7). In this case, a set element is depicted as an icon and it is represented by its position, whereas a set is depicted as a circle and it is represented by the coordinates of its center and any point on the circumference.

It is worth noting that although other features could be considered as syntactic attributes (for example, a circle could be syntactically described by the coordinates of its center and the measure of its radius), the use of the coordinates of points in the Cartesian plane makes the definition of geometric-based classes independent on the particular shape of the graphical objects.

Table 4 summarizes the descriptions of the basic geometric-based classes of visual languages.

3.2.1. Class String

This class is suitable for modeling all the string languages. It can be seen as the reduction of a geometric class to the linear case. The graphical objects of a string language are textual characters[b].

Syntactic attributes

The only syntactic attribute of a character is an integer denoting its *position* within a string.

Relations

The only relation is the *string concatenation*, denoted by CONC and defined as follows.

[a] The term *concatenation* as used in this paper refers to any spatial arrangement of graphical objects not intersecting their areas.

[b] For sake of uniformity, in the class string characters are considered as graphical objects.

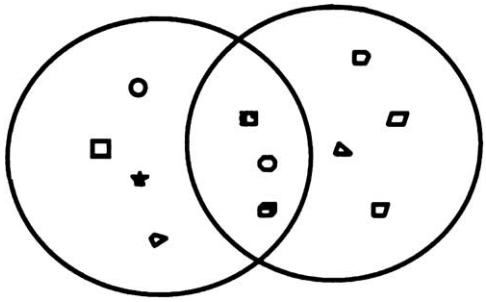

Figure 7. A sample Venn diagram

Table 4. Basic geometric-based classes of visual languages

Class	Relations	Syntatic attributes
String	String concatenation	The position of the symbol in the string
Iconic	Any spatial concatenation and overlapping	The position of the icon in the Cartesian plane
Box	Any spatial composition	The upper-left and the lower-right points of the box

Definition 3. Given the characters a and b with positions i and j, respectively, then a CONC b holds if and only if $j = i + 1$.

3.2.2. Classic Iconic

The graphical objects of the class iconic are icons. An icon is conceived as an image within a box of fixed size.

Syntactic attributes

Each icon has one syntactic attribute, which is a pair of integers denoting the coordinates of the icon in a grid of cells of the same size as an icon bounding box.

Relations

The relations are *iconic spatial compositions*, defined as follows.

Definition 4. An iconic spatial composition REL consists of a set of pairs of integers (m, n). Given a relation REL and two icons a and b with syntactic attributes (x, y) and (h, k), respectively, then a REL b holds if and only if $\exists\ (m, n) \in$ REL such that $h = x + m$ and $k = y + n$.

The spatial arrangements produced by such compositions represent either spatial concatenations or overlapping. In particular, the overlap of two icons is denoted by the relation Overlap$\equiv\{(0,0)\}$. Other examples of spatial concatenations follow:

$$\text{Right-step} \equiv \{(1, 0)\}$$

$$\text{NorthEast-step} \equiv \{(1, 1)\}$$

$$\text{Right} \equiv \{(m, n) | m > 0\}$$

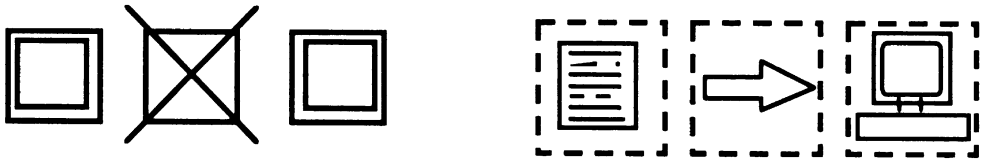

Figure 8. Two iconic sentences

Figure 8 shows two simple iconic sentences: the first one, taken from the Heidelberg Icon Set [15], describes the operation 'delete string', while the second one, taken from [5], represents the operation 'display a text file'. In both cases the icons are related through the *right-step* spatial concatenation.

3.2.3. Class Box

The graphical objects of this class are characterized by their bounding boxes, i.e. they can be syntactically described and manipulated through their bounding box, whatever the shape of the object is. The main difference between the class box and the class iconic is that the former allows bounding boxes with variable size. This leads to different definitions of syntactic attributes and relations within the two classes.

Syntactic attributes

Each graphical object has two syntactic attributes which are pairs of integers denoting the coordinates of the upper-left and lower-right points of its bounding box.

Relations

The relations are *box spatial compositions*, which generalize the iconic compositions. However, unlike the iconic case, a box spatial composition cannot be defined independently from the syntactic attributes of the first argument, as this defines an area of variable size. In order to define a box spatial composition REL with respect to a box a, we use two functions $\mathrm{UL}_{\mathrm{REL}}(m, n, m_1, n_1)$ and $\mathrm{LR}_{\mathrm{REL}}(m, n, m_1, n_1)$ that map the coordinates of the upper-left (m, n) and lower-right (m_1, n_1) points of the bounding box of a onto sets of points.

Definition 5. Given a box a with syntactic attributes (x, y) and (x_1, y_1) and a box b with syntactic attributes (h, k) and (h_1, k_1), then a REL b holds iff $(h, k) \in \mathrm{UL}_{\mathrm{REL}}(x, y, x_1, y_1)$ and $(h_1, k_1) \in \mathrm{LR}_{\mathrm{REL}}(x, y, x_1, y_1)$.

In other words, a REL b holds if and only if the upper left and the lower right points of b are contained respectively in the two areas calculated on the syntactic attributes of a through the functions $\mathrm{UL}_{\mathrm{REL}}$ and $\mathrm{LR}_{\mathrm{REL}}$ (see Figure 9).

The above definition of composition yields three types of spatial arrangements: inclusion, intersection and spatial concatenation. As an example, given a box a with syntactic attributes (x, y) and (x_1, y_1), the relations INCLUDE, INTERSECT and the spatial concatenations RIGHT and DOWN with respect to a are defined below.

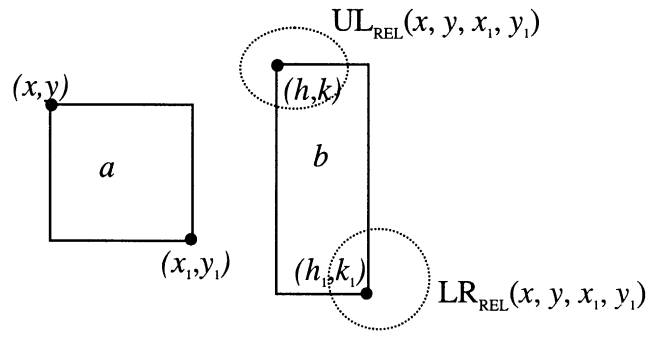

Figure 9. Upper-left (UL) and lower-right (LR) constraints of a box relation REL

INCLUDE

The relation *a* INCLUDE *b* holds if the bounding box of *b* is contained in the bounding box of *a*:

$$\mathrm{UL}_{\mathrm{INCLUDE}}(x, y, x_1, y_1) = \{(m, n) | x \leq m < x_1,\ y_1 < n \leq y\}$$

$$\mathrm{LR}_{\mathrm{INCLUDE}}(x, y, x_1, y_1) = \{(m, n) | x < m \leq x_1,\ y_1 \leq n < y\}$$

It is worth noting that this definition includes the case when the boxes *a* and *b* overlap.

INTERSECT

The relation *a* INTERSECT *b* holds if the bounding boxes of *a* and *b* have a nonempty intersection:

$$\mathrm{UL}_{\mathrm{INTERSECT}}(x, y, x_1, y_1) = \{(m, n) | m \leq x_1, n \geq y_1\}$$

$$\mathrm{LR}_{\mathrm{INTERSECT}}(x, y, x_1, y_1) = \{(m, n) | m \geq x,\ n \leq y\}$$

It is worth noting that if *a* INCLUDE *b* holds, then also *a* INTEREST *b* holds.

RIGHT

The relation *a* RIGHT *b* holds if the upper-left point of the bounding box of *b* is on the right of the right edge of the bounding box of *a*:

$$\mathrm{UL}_{\mathrm{RIGHT}}(x, y, x_1, y_1) = \{(m, n) | m > x_1\}$$

DOWN

The relation *a* DOWN *b* holds if the upper-left point of the bounding box of *b* is below the lower edge of the bounding box of *a*:

$$\mathrm{UL}_{\mathrm{DOWN}}(x, y, x_1, y_1) = \{(m, n) | n < y_1\}$$

The functions $\mathrm{LR}_{\mathrm{RIGHT}}$ and $\mathrm{LR}_{\mathrm{DOWN}}$ can be left unspecified in the definition of the relations RIGHT and DOWN.

As an example, Figure 10 shows a sample box visual sentence. The relation INTERSECT holds between the boxes *b* and *c*; these boxes are both included in the box *a*; finally, the relations RIGHT and DOWN hold between the box *a* and the boxes *e* and *d*, respectively.

Well-formed two-dimensional arithmetic expressions can be modeled according to this class. As a matter of fact, each expression can always be modeled as a set of

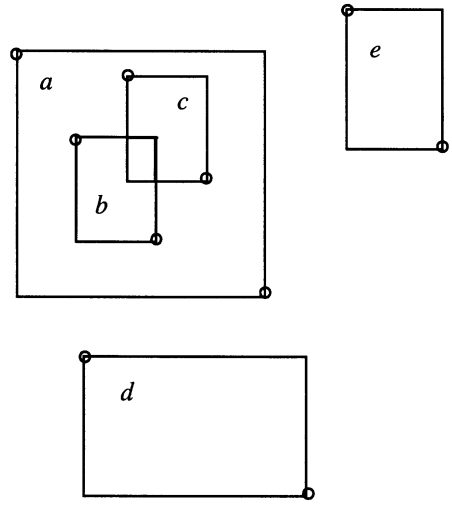

Figure 10. A box visual sentence

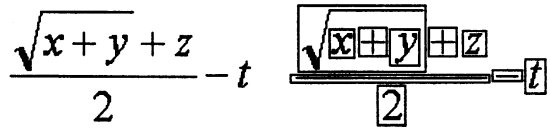

Figure 11. An arithmetic expression and its box structure

boxes which are composed according to the spatial concatenation and the inclusion relations. As an example, an arithmetic expression and its box decomposition are shown in Figure 11.

$$\frac{\sqrt{x+y}+z}{2}-t$$

3.3. An Inclusion Hierarchy of Visual language Classes

The classes of visual languages described in the previous subsections can be represented in an inclusion hierarchy that expresses the generalization–specialization relation between them, as shown in Figure 12.

Abstract classes are written in italics. The most general connection-based class in graph, whereas the most general geometric-based class is Box. The Class plex can be seen as a specialization of the class graph where attaching regions contain single points; the class Iconic specializes the more general class box (icons can be represented by boxes of fixed size), while the class String can be seen as the specialization of the class iconic to the linear case.

3.4. Hybrid Classes of Visual Languages

A visual language can be modeled by using one of the basic classes defined above or by integrating features from more of them. Indeed, a visual language may present graphical

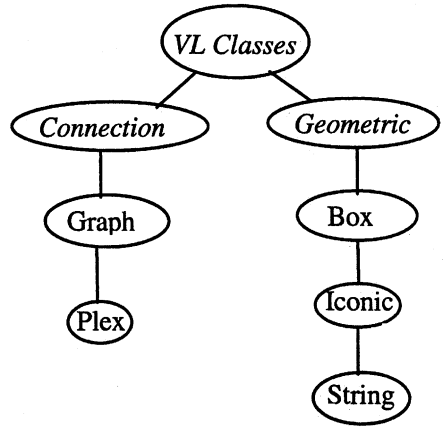

Figure 12. VL classes inclusion hierarchy

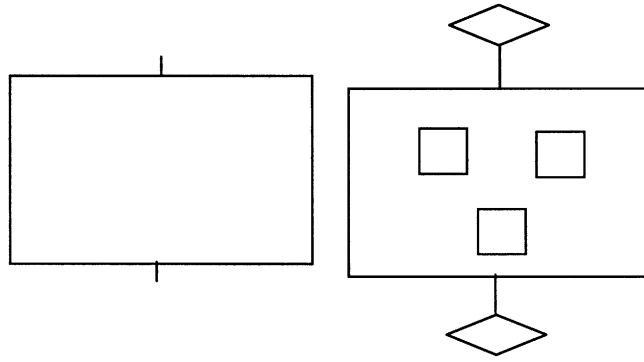

Figure 13. A graphical object and its use in hybrid modeling

objects with properties derived from different basic classes. As an example, let us consider a graphical object with one input and one output port and that might contain other graphical objects (see Figure 13). This object can be modeled by assigning it the upper left and lower right points of its bounding box and an attaching point for each port as syntactic attributes. In this case we say that the graphical object has been modeled in a *hybrid* way according to the classes box and plex.

The concept of hybrid modeling allows to capture and extend the description capabilities of the basic classes of our framework. The introduction of this modeling paradigm is motivated by the necessity to support the definition of complex visual languages, including many practical visual languages like Statecharts, UML diagrams, etc. As an example, let us consider the graphical object 1 of the Statechart sentence shown in Figure 14. Intuitively, such an object needs the syntactic attributes defined in both the classes graph and box. The former should be used to express link relationships, such as connections between states, whereas the latter should allow to define spatial relationships such as the containment between states and superstates.

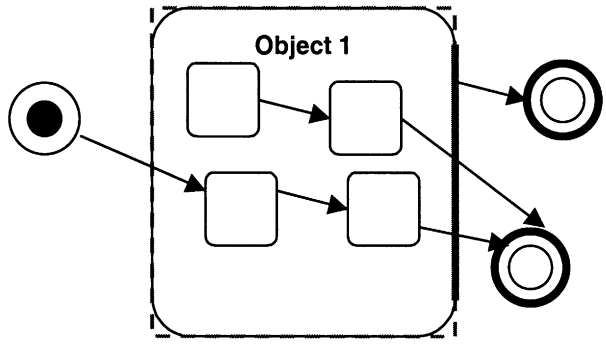

Figure 14. A statechart sentence

Another important issue to be taken into account when designing complex visual languages is the concept of hierarchical languages, that is, visual languages whose graphical objects can be annotated by visual sentences from possibly different visual languages. To this aim, in addition to the specific relations, a class may include a special relation that associates a visual sentence to a graphical object. In the following, we will refer to this type of relation as the *annotate* relation. As an example, in the flowchart in Figure 1 the *annotate* relation can be used to associate each node with a piece of code describing the action of condition to be executed. The *annotate* relation should not be confused with the *include* relation of the class box, since it establishes a relationship that is logical rather than spatial. Moreover, when used recursively, the *annotate* relation enables the specification of nested visual languages

4. Classification of Existing Visual Languages

In this section we provide an overview of visual languages existing in the literature. We have also given a possible classification for them according to our framework of classes.

4.1. Graph

The class graph may be used to describe the structure of a number of graph languages. Many of these languages have been used within software engineering methodologies. Examples include languages based on data flow graphs, state transition diagrams, Petri nets, entity–relationship diagrams, SADT diagrams, Class and Object diagrams.

In data flow graph languages the operations are typically associated to boxes and the data flow along the arrows connecting them. In general, graphical objects of such languages may present a variable number of attaching points (i.e. attaching regions), as in the case of Data Flow Diagrams used for specifying the data flow in software systems, or in the case of PegaSys [16] which is a diagrammatic system designed to provide a programming environment where all steps of software life cycle are supported by graphical interaction languages. Phred [17] is a visual parallel programming environment that allows a software designer to create Phred programs, to statically analyze them for determinacy, and to interpret them. A Phred program is composed of a control graph, a data

flowgraph, and a set of node interpretations represented by procedure specification (written in C language).

Transition diagrams of finite state automata are another typical example of graph languages. The graphical objects of such languages are nodes and arcs representing the states and the transitions of the automaton, respectively. The attaching region of a node is given by its circumference, while an arc has two attaching points corresponding to its end points. An example of language based on state transition diagrams has been proposed by Jacob [18], and it is used for specifying user interfaces in a graphical manner.

SADT [19] diagrams have been used in several water-fall software engineering methodologies to specify the functional part of a system, which has often to be integrated with the data specification. SADT diagrams are composed of rectangular boxes, with arrows entering left and upper sides, and arrows leaving right and bottom sides. Boxes represent activities of processes, arrows entering the left side represent the activity's inputs, whereas those leaving the right side represent its outputs. The activity is carried out by one or more actors, which are represented through arrows entering the box from the bottom side. The actors can be human, software systems, etc. Moreover, the activity has to be carried out according to constraints that are represented through arrows entering the upper side. The outputs of a box can be input to another box representing a successive activity. In order to face scale-up problems, SADT can be specified in a hierarchical fashion. In other words, a macro-activity A can be detailed by annotating its corresponding box with another SADT diagram with boxes $a_1, a_2, \ldots, a_n$ representing the micro-activities composing A. Obviously, the hierarchical decomposition must obey to some constraints, such as the inputs and outputs of A must correspond to inputs and outputs of some box in the detailed SADT of A.

A sample of SADT diagram is depicted in Figure 15.

Many visual programming systems are based on the Petri nets formalism for specifying multimedia, concurrent, and real-time systems. For example, the MOPS-2 system [20] uses colored Petri nets to allow parallel systems to be constructed and stimulated in a visual manner. The VERDI system [21] uses a form of Petri nets for specifying and simulating distributed systems: the specification is animated by moving tokens around the network. Cabernet [22] is a visual environment based on high-level Petri nets [23] designed to support the specification and verification of real-time systems. An example of graph language used to specify multimedia applications is given by the teleaction object (TAO) [24]. A sentence from this language is shown in Figure 16. There graph nodes

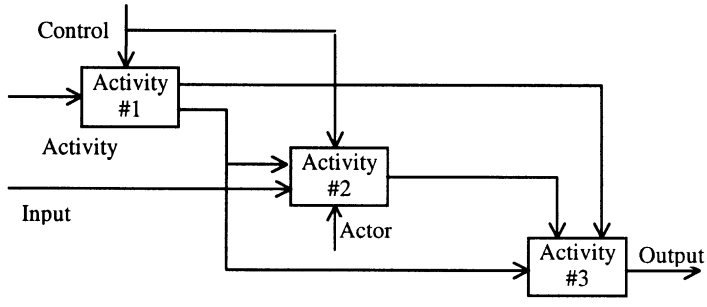

Figure 15. The structure of an SADT diagram

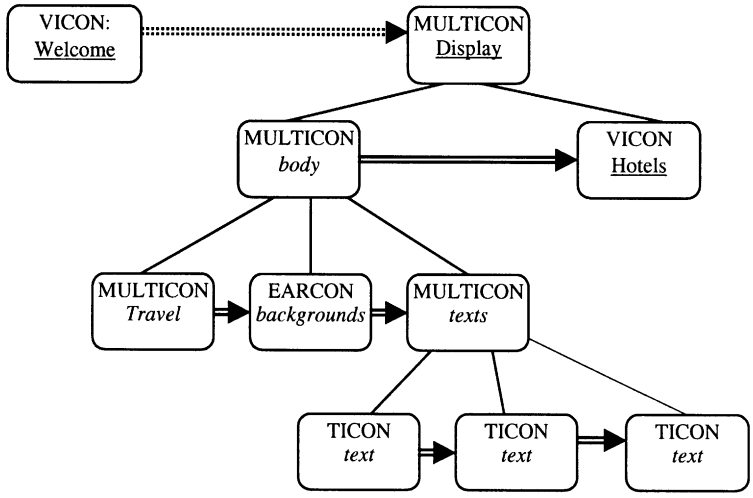

Figure 16. The structure of a Tao hypergraph

represent text, image, video, motion, or composed multimedia objects, and are interconnected through synchronization, attachment, spatial, or annotation links.

4.2. Plex

The class plex can be used to describe the structure of visual languages like flowcharts, chemical structures, Boolean and electric circuits, and so on. For instance, in flowchart languages each graphical object (boxes, diamonds, etc.) has a pre-defined number of attaching points (two for boxes, three for diamonds, etc.), and graphical objects can be connected only through links visualized as polylines. Examples of flowchart-based languages include (First Programming Language) (FPL), [25], particularly suited to help novices in learning programming since it eliminates syntactic errors, OPAL [26] that allows doctors to enter knowledge about cancer treatments into an expert system, and Pict [27] that uses flowcharts with boxes annotated by color pictures instead of text.

Also data flow graph languages whose boxes have a fixed number of attaching points can be modeled according to the plex syntax. This is in case, for example, of PROGRAPH [28] which is a structured, functional language introduced to solve comprehension problems of conventional textual functional languages, and InterCONS [29] which is a data flow system developed in Smalltalk that supports programming by example.

Nested data flow graph languages can be also modeled according to the class plex, where the nesting is obtained by using the relation annotate. Usually, in a nested data-flow graph each node represents an operator (function), an operand (variable), or another directed graph represented by an iconic name. Examples include the programming by example system Fabrik [30] developed in smalltalk and used for constructing user interfaces (see Figure 17 for an example of sentence in Fabrik) and the commercial product Lab-VIEW[c] running on Macintosh and used for controlling external devices. Lab-VIEW provides procedural abstraction, control structures, and many useful primitive

[c] National Instruments, LabVIEW, 12109 Technology Blv, Austin, Texas, 78727.

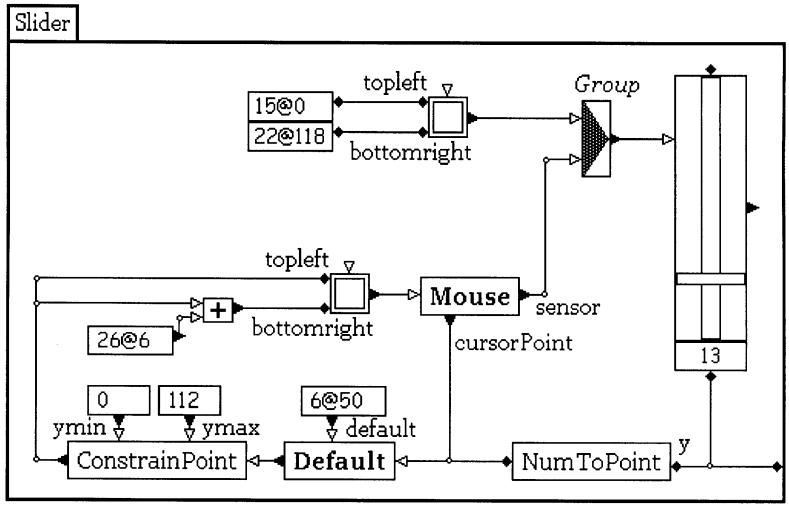

Figure 17. An example of Fabrik diagram

components such as knobs, switches, and mathematical functions. Show and Tell [31,32] is a visual programming language for defining interfaces of relational databases (scheme and queries). A Show and Tell database scheme is defined by a two-dimensional layout of *base boxes*, while a query is defined by a data flow program.

Other general-purpose nested data flow visual programming language systems are Hyerflow [33] and its derivatives ProtoHyperflow [34] and HF/PP [35]. The syntax of these languages consists of boxes and arrows, a box representing a process, and an arrow representing a data flow between processes. Boxes may contain data flowgraphs for defining new functions and to build conditionals. Hyperflow and HF/PP are designed as visual languages for a pen-based multimedia system, while ProtoHyperflow is implemented on a traditional mouse/CRT-based system.

Another visual language that can be modeled by the class plex has been proposed by Karsai [36]. It consists of a configurable visual programming environment that can be customized for various application domains. Also the *BLOX* languages proposed by Glinert [37] are plex languages. Visual sentences are composed by joining graphical objects using the usual jigsaw-puzzle 'lock and key' metaphor to plug protrusions ('knobs') into correspondingly shaped indentations ('sockets') so that the two justaposed tiles interlock. Moreover, BLOX substructures can be encapsulated (nested) into BLOX elements. Languages for representing logical circuits and structured programs (in a similar way to nested flowchart, where function boxes may hide sub-flowcharts) have been defined using the BLOX methodology.

4.3. Box

The class box can be used to describe high-level interface languages for geographical information systems. For example, Cigales [38] is a graphical query language based upon spatial query by example. The basic idea of this language is to express a query by drawing a pattern according to the user's mental model of the data to be retrieved. The graphical objects of this language are geometric objects, such as lines and areas, while queries

(sentences) are composed by means of spatial rules, such as inclusion, intersection, and adjacency.

A typical example of box language is given by the Nassi–Schneidermann diagrams [39] used to express computer programs. In this formalism each program is associated to a box. The boxes also delimit program blocks. Special box notations are dedicated to control statements. As an example, an IF statement is expressed by using a box divided in several sections: an upper triangle contains the test condition, whereas the rest of the box is divided into columns, each containing a possible result for the test condition and the associated code. An example of program expressed through the Nassi–Schneidermann notation is given in Figure 18. Notice that the nesting of programs is realized through the relation annotate.

Another example of box visual language is given in [40], where a visual calendar is used to specify authorization policies in task based environments (see Figure 19). Each calendar cell may include one or two authorizations. In the case it contains two of them, these are considered as right-adiacent.

4.4. Iconic

A number of visual languages have been proposed in the literature which may be modeled according to the class iconic. For example, HI-VISUAL [41] provides an iconic

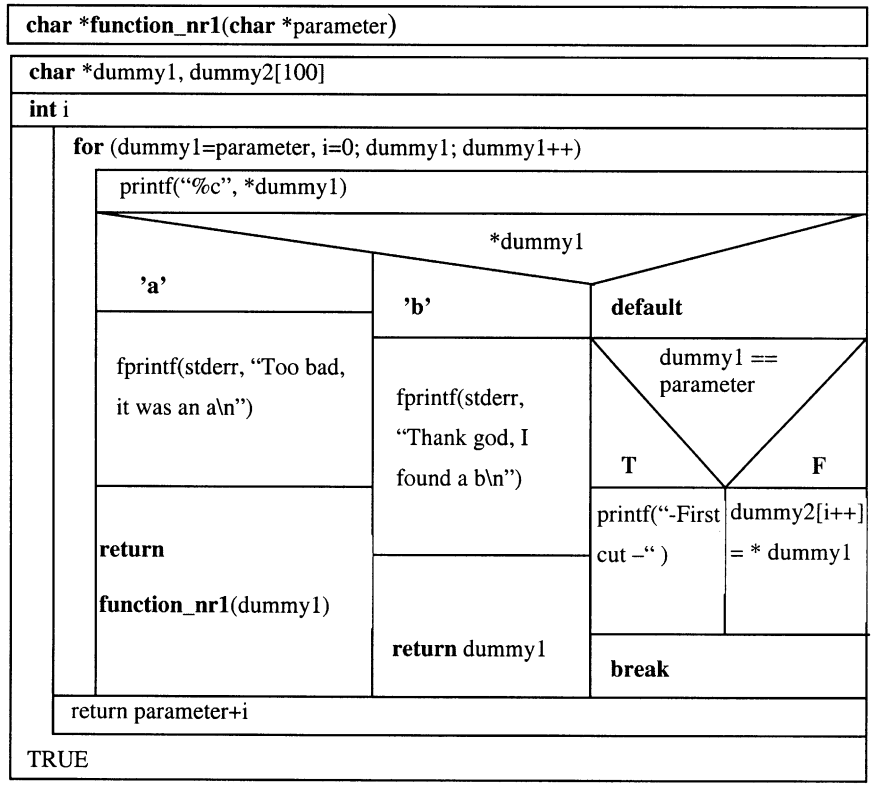

Figure 18. A Nassi-Schneidermann diagram

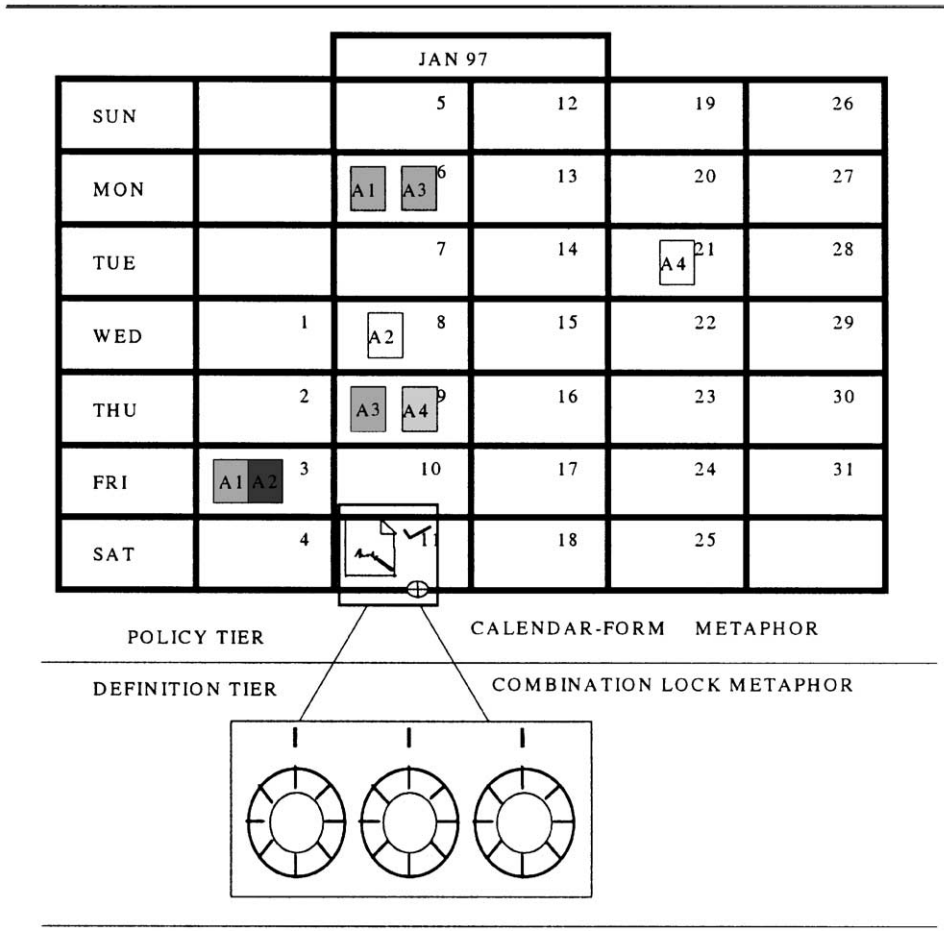

Figure 19. A visual language for specifying security policy

framework to construct data flow programs by composing icons in a two-dimensional display screen according to pre-defined rules. Media Streams [42] is an iconic language system which enables users to create multi-layered iconic annotations of streams video data. The SIL-ICON [43] compiler is a system for specifying, interpreting and prototyping icon-oriented systems. It uses context-free grammar augmented with spatial operators to define a visual language and allow iconic sentences to be constructed by arranging icons in a two-dimensional fashion and parsed and interpreted according to the specification rules of the language. Minspeak [44] is an iconic language system used in augmentative communication by people with speech disabilities. It consists of multi-meaning one-dimension iconic sentences which are used to retrieve messages (words or word sequences) stored in the memory of a microcomputer. A built-in speech synthesizer allows to generate the voice output. An example of iconic sentence of Minspeak representing the concept *fear* is shown in Figure 20.

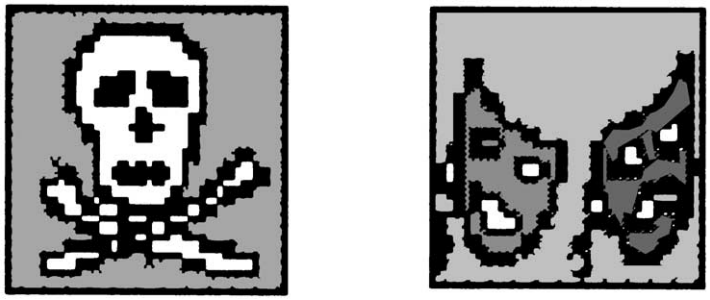

Figure 20. An iconic sentence of Minspeak

4.5. Hybrid

Box-and-graph visual languages are an example of hybrid modeling obtained by grouping features of the classes box and graph. A graphical object has both the upper-left and lower-right points of its bounding box and a pre-defined number of attaching points and/or regions as syntactic attributes. The relations that can be composed to form visual sentences are both geometric (spatial inclusion, horizontal and vertical concatenation, etc.) and connection based. Statecharts [12, 45] are an example of language that can be modeled as box-and-graph. Statecharts are an extension of state transition diagrams introduced to specify large and complex reactive systems, that is, event-driven continuously reacting to stimuli systems.

The main features of statechart transition diagrams are AND/OR decomposition of states, inter-level transitions, an a broadcast mechanism for communication between concurrent components. Graphical objects representing states or AND/OR decomposition of states may contain nested statecharts. Examples of visual languages based on statechart formalism include Mirò [46] which is a language for defining security constraints (in particular user accesses to files) in operating systems, and StateMaster [47] which is a visual system for programming graphical user interfaces.

The Unified Modeling Language (UML) [48] is a set of graphical languages used for specifying and modeling systems. It represents a widely accepted standard in the context of object-oriented software engineering methodologies.

An example of class diagram is given in Figure 21. It shows a number of classes connected to the class *Pump* through the *Composition* relation. The state diagram for the class *Pump* is shown in Figure 22. This can be represented in our framework by using the relation annotate between the class *Pump* and its associated state diagram. Notice that State Diagrams in UML are hierarchical, meaning that each state can have nested State Diagrams to describe the details of its dynamic behavior.

The UML diagrams can be modeled by using they hybrid paradigm. In fact, although the graph component is predominant in their structure, there are some plex and box features in them. For example, there are cases in which arrows cannot be simply modeled as interconnection relations, but they need to be represented as tokens with fixed attaching points corresponding to their start and end points. This is the case of the arrows for specifying composition relations in UML Class diagrams. Moreover, UML state diagrams present some box features. In fact, as said above they can have macro-states whose behavior is detailed through another state diagram. These sentences can be visually modeled through boxes including inner state transition diagrams.

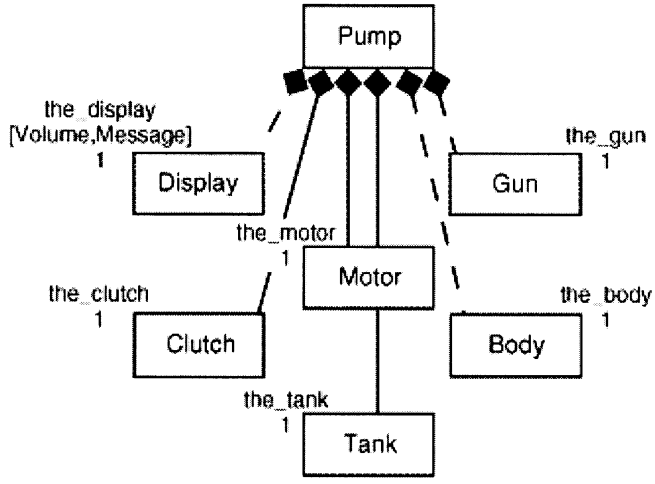

Figure 21. An example of class from a class diagram.

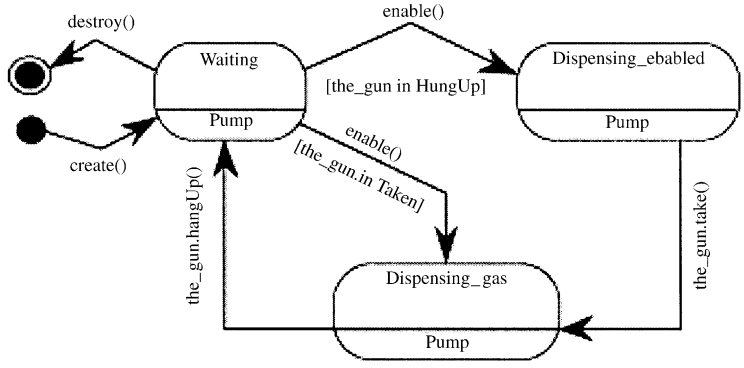

Figure 22. The State Diagram for the class Pump of Figure 21

5. The VLCC System

In this section, we show how to implement a visual language according to a given class by using the Visual Language Compiler-Compiler (VLCC) system [3]. VLCC is a tool for the automatic generation of visual programming environments implementing visual languages. A prototype of the VLCC system has been implemented using object-oriented technology under MSWindows98. The VLCC supports the language designer in the definition of a visual language by assisting him/her in the specification of the graphical objects, the syntax and the semantics of the language.

The main components of VLCC are the *Grammar Editor* (GE) and the *Visual Programming Environment Generator* (VPEG) (see Figure 23). GE supports the language designer during the specification of the visual aspects (the graphics) and the logical features (the syntactic attributes) of the language terminals, both accomplished through a *Symbol Editor*. Moreover, GE allows the designer to define the syntax and the semantics of the visual language, either textually by entering a YACC-like specification through a

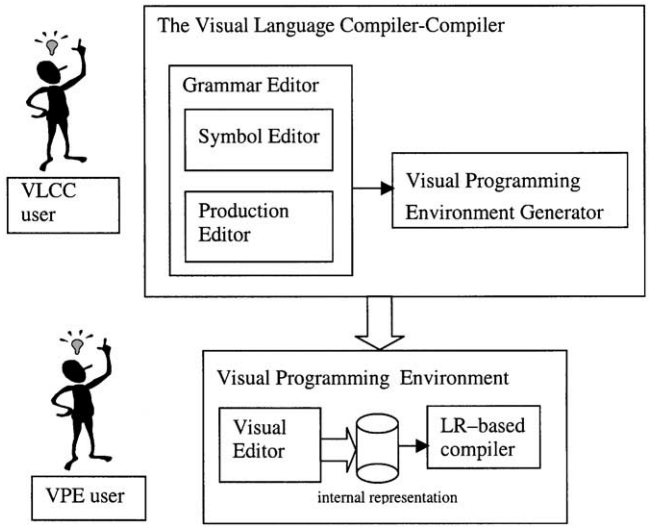

Figure 23. The VLCC architecture

text editor or graphically through a *Visual Production Editor*. Starting from the supplied language specification, VPEG generates an integrated visual programming environment consisting of a graphical editor for the definition of the visual sentences and a compiler for analyzing and translating them. Each new VLCC generated environment is structured in three levels:

- the graphical interface,
- the visual language class implementation,
- the language compiler and tokens.

Each level is implemented by a set of C++ classes that inherit or use classes from the higher levels. The first level is common to all the visual language environments generated by VLCC: it mainly contains abstract C++ classes implementing the general skeleton of the graphical editor. As an example, this module includes virtual functions defining the way the graphical editor draws a token, and functions implementing command menus such as copy, paste, undo, etc.

The second level implements a given visual language class, and it is shared among all the VLCC environments generated for visual languages modeled according to that class. This module contains classes implementing the structure of visual sentences (i.e. the syntactic attributes and the feasible relations). The current version of VLCC implements all of the classes of the framework presented in this paper. However, the object-oriented architecture of the system makes it easy to add implementations of newly defined classes. So far the VLCC systems has been used to implement several practical visual languages, like finite state automata, semi-structured flowcharts, UML diagrams, etc.

The third level is specific to each visual language and contains C++ classes implementing the compiler and tokens for that language. The compiler is based on an efficient LR-like methodology [4]. A token contains information regarding both its graphical aspect and the association between its syntactic attributes and its image. Moreover, the

C++ class implementing a token inherits all the functions for its graphical manipulation from the above levels.

While the first two levels are pre-defined and implemented in dynamic libraries, and in the third level is built by a visual language designer through the use of the VLCC *Grammar Editor*. Finally, the executable code for the visual programming environment

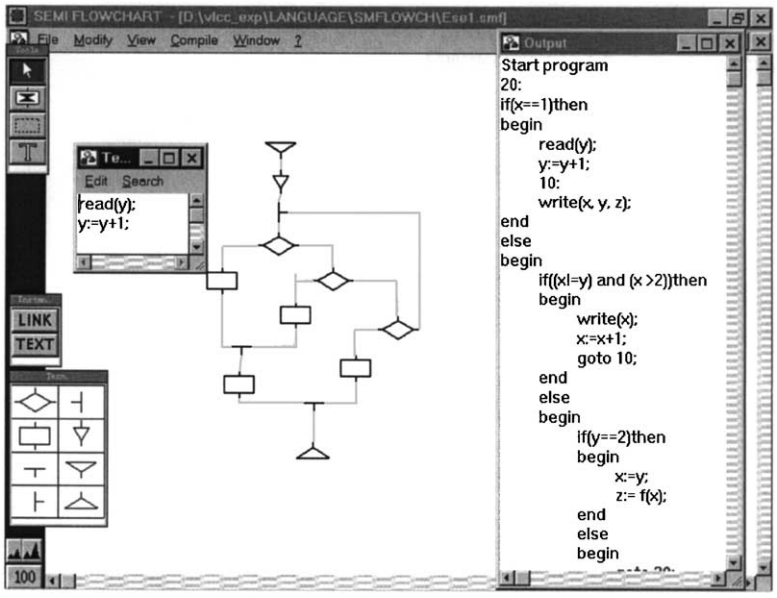

Figure 24. The VLCC generated Flow-Chart environment

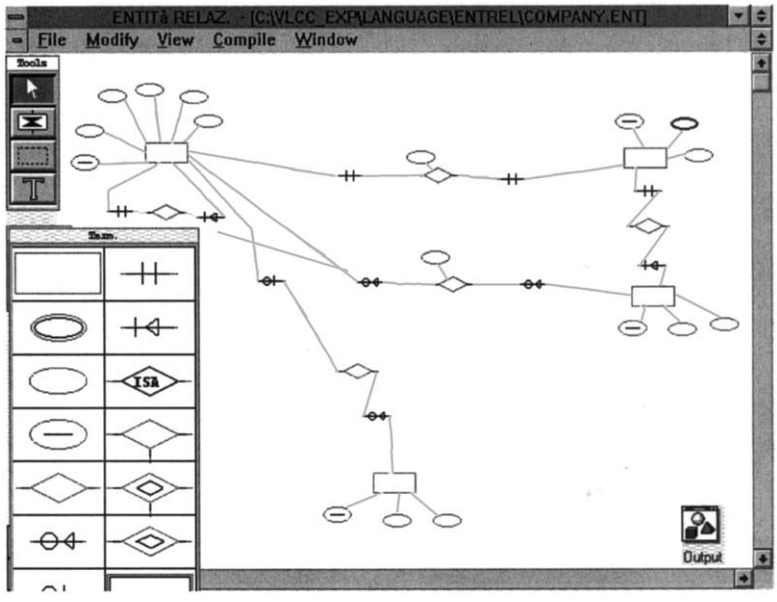

Figure 25. The VLCC generated ER environment

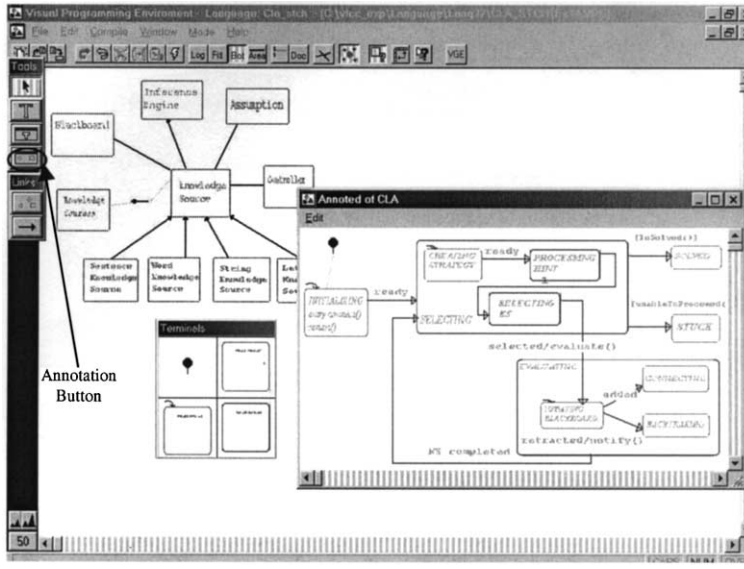

Figure 26. The VLCC generated UML environment

is obtained by compiling the C++ code implementing the third level with the pre-defined libraries.

The generated graphical environment allows a user to draw and compile visual sentences through two palettes, one used to set the language graphical objects on the screen and one to instantiate their syntactic attributes. The latter also includes button to annotate graphical objects with textual or visual sentences.

As an example, Figures 24–26 show three VLCC-generated visual programming environments implementing semi-structured flowcharts (Plex), Entity–Relationship (ER) diagrams (Graph), and UML diagrams (Hybrid), respectively.

Figure 24 also shows the result of the compilation process, represented by the Pascal code corresponding to the flow-chart, which is shown in the output window to the right of the main window. Similarly, the ER diagram of Figure 25 is compiled into the SQL code shown in Figure 27. Finally, the compiler for the UML environment checks the syntactic and semantic correctness of the diagrams and can produce either intermediate UML design diagrams or the skeleton of the final C++ code for the system being modeled.

6. Concluding Remarks

In this paper we have presented a classification framework for visual languages, and the VLCC system as a visual language compiler generator based upon the framework. We believe that this framework can play an important role in the design of visual languages. In particular, it provides a visual language designer with guidelines to analyze the characteristics of the language, and to associate it to an appropriate class. In this way, the designer can exploit the pre-defined design framework of the selected class, and the customized visual design environments of VLCC providing specific support for

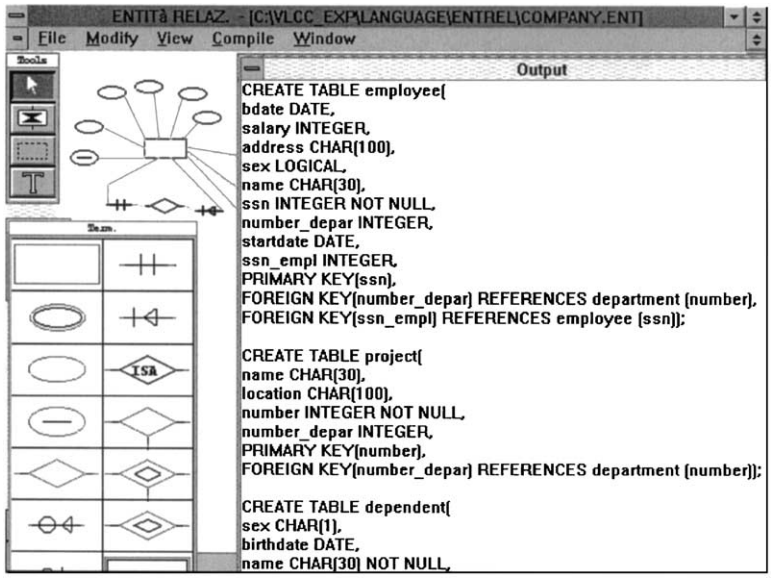

Figure 27. The SQL code corresponding to the ER diagram in Figure 25

the visual languages of the chosen class. We have also provided a wide characterization of existing visual languages by modeling each of them through what we feel to be the most appropriate class from the framework.

It is worth noting that enhancing our framework through hybrid and hierarchical paradigms opens the way to model any visual language. In fact, no matter how complex the visual language is, we can always derive an hybrid class to model it. This might also require adding new specific classes to the framework. For example, in order to model Contour Maps we have to extend the class *Geometric* to derive a new class having graphical objects of different geometric shapes, modeled through a vector of points.

Several other visual language tools have been recently proposed which provides templates for classes of visual languages. A recent proposal refers to *category theory* to characterize families of connection-based languages in terms of morphims among elements [49]. Moreover, categories of languages were defined according to an Entity–Relationship approach in [50].

Metamodel approaches are gaining interest in the visual language community, following their success in defining the visual languages of UML. A metamodel approach is implicit in most generators of diagrammatic editors. Most such generators are based on the translation of some formal syntax into a set of procedures controlling user interaction, and on constructing the semantic interpretation of the diagram in a parsing process. Examples of such tools where the formal syntax is some variant of graph rewriting are Diagen [51] based on hypergraph rewriting, and GenGEd [52].

In MetaBuilder [53], a designer defines a new visual language by drawing its metamodel class diagram, from which an editor is automatically generated. MetaBuilder does not provide support for the management of textual constraints, nor does it define basic metamodels for families of diagrammatic languages.

The GenIAL tool [54] allows the definition of connection-based languages starting from two basic classes: *Entity* and *Connection*. A designer specializes *Connections* by defining their arities and orientations. Moreover, attributes taking values in a set of basic primitive types can be associated with each token, thus defining classes of visual terms. Rules of a visual rewriting systems are interactively defined in accordance with the class of each term; conditions and a global check are expressed a textual form. An incremental editor is automatically generated from the specification.

In future we aim to use this design methodology for designing other practical visual languages. This will let us define new hybrid classes within VLCC and to further specialize its parametric tools. As we increase specialization within the framework and in VLCC, this should yield faster prototyping of complex visual languages. The uniform and fast prototyping of visual languages can also provide benefits to the construction of metacase environments. In fact, although visual language compiler generators promise to be suitable metacase environments, they have not been widely used to this aim due to the complex visual language design process to be undergone for generating CASE tools. The use of our predefined design framework and class-specification design tools can considerably improve this process. Moreover, multimedia and web engineering are causing a proliferation of design methodologies. Since these fields require adaptable methodologies and tools, we would like to model and implement visual languages of emerging web and multimedia engineering methodologies within our framework.

Another appealing use of our framework is in the context of multimedia databases. In particular, other than using the framework to construct database design notations, the framework can be used to design visual query languages. In fact, the complex nature of multimedia data has encouraged researchers to exploit visual query languages for querying image, sound and video based upon their content. Moreover, we would like to investigate how to manage features like gestures and time constraints within our frame work, to enable the modeling the dynamic visual languages, which will broaden the range of applications.

References

1. S. K. Chang, G. Costagliola, G. Pacini, G. Tortora, M. Tucci, B. Yu & J. S. Yu (1995) A visual language system for user interfaces. *IEEE Software* **12**, 33–44.
2. S. S. Chok & K. Marriot (1995) Automatic construction of user interfaces from constraint multiset grammers. *Proceedings of IEEE Symposium on Visual Languages*, September 1995, pp. 242–249.
3. G. Costagliola, G. Tortora, S. Orefice & A. De Lucia (1995) Automatic generation of visual programming environments. *IEEE Computer* **28**, 56–66.
4. G. Costagliola, G. Tortora, S. Orefice & A. De Lucia (1997) A parsing methodology for the implementation of visual systems. *IEEE Transactions on Software Engineering* **23**, 777–799.
5. C. Crimi, A. Guercio, G. Pacini, G. Tortora & M. Tucci (1990) Automating visual language generation. *IEEE Transactions on Software Engineering* **161**, 1122–1135.
6. E. J. Golin & T. Magliery (1993) A compiler generator for visual languages. *Proceedings of IEEE Workshop on Visual Languages*, Bergen, Norway, pp. 314–321.
7. N. Goldman & R. Balzer (1999) The ISI visual design editor generator. *Proceedings of IEEE Symposium on Visual Languages*, Tokyo, Japan, September 1999, pp. 20–27.
8. Special Issue on Visual Programming. *IEEE Computer*. **28**.
9. D. Zhang & K. Zhang (1999) VisPro: a visual language generation toolset. *Proceedings of IEEE Symposium on Visual Languages*, Halifax, Canada, pp. 195–202.

10. K. Marriott & B. Meyer, (1998) The CCMG visual language hierarchy. In: *Visual Language Theory* (K. Marriot & B. Meyer, ed.) Springer, Berlin.
11. K. Marriott, B. Meyer & K. Wittenburg (1998) A survey of visual language specification and recognition. In: *Visual Language Theory* (K. Marriot & B. Meyer, ed.). Springer, Berlin.
12. D. Harel (1987) Statecharts: a visual formalism for complex system. *Science of Computer Programming* **8,** 231–274.
13. S. C. Johnson (1975) YACC: Yet another compiler compiler. Tech. Report 32, AT&T Bell Laboratories, Murray Hills, NJ.
14. J. Feder (1971) Plex languages. *Information Science* **3**, 225–241.
15. G. Rohr (1986) Using visual concepts. In: *Visual Languages* (S. K. Chang, T. Ichikawa & P. Ligomenides, eds). Plenum, New York.
16. M. Moriconi & D. F. Hare (1985) Visualizing program design through PegaSys. *IEEE Computer* **18**, 72–85.
17. A. Beguelin & G. Nutt (1994) Visual parallel programming and determinacy: a language specification, an analysis technique, and a programming tool. *Journal of Parallel and Distributed Computing* **22**, 235–250.
18. R. J. K. Jacob (1985) A state transition diagram language for visual programming. *IEEE Computer* **18**, 51–59.
19. D. T. Ross & K. E. Schoman Jr (1977) Structured analysis for requirement definition. *IEEE Transactions on Software Engineering* **3,** 6–15.
20. T. Ae, M. Yamashita, W. C. Cunha & H. Matsumoto (1986) Visual user-interface of a programming system: MOPS-2. *Proceedings of IEEE Workshop on Visual Languages*, Dallas, TX. IEEE CS Press, Silver Spring, MD, pp. 44–53.
21. M. Graf, (1987) A visual environment for the design of distributed systems. *Proceedings of IEEE Workshop on Visual Languages*, Linkoping, Sweden. IEEE CS Press, Silver Spring, MD, pp. 330–344.
22. C. Ghezzi & M. Pezzè (1992) Cabernet: an environment for the specification and verification of real-time systems. *1992 DECUS Europe Symposium*, Cannes, France, 7–11 September.
23. G. Ghezzi, D. Mandrioli, S. Morasca & M. Pezzè (1991) A unified high-level Petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering* **SE-17**.
24. S. K. Chang (1996) Extending visual languages of multimedia. *IEEE Multimedia* **3**, 18–26.
25. N. Cunniff, R. P. Taylor & J. B. Black (1986) Does programming language affect the type of conceptual bugs in begineers' programs? A comparison of APL and Pascal. *Proceedings of SIGCHI'86: Human Factors in Computing Systems*, Boston, MA, pp. 175–182.
26. M. A. Musen, L. M. Fagen & E. H. Shortliffe (1986) Graphical specification of procedural knowledge for an expert system. *Proceedings of IEEE Workshop on Visual Languages*, Dallas, Texas, pp. 167–178.
27. E. P. Glinert & L. Tanimoto (1984) Pict: an interactive graphical programming environment. *IEEE Computer* **17**, 7–25.
28. T. Pietrzykowsky, S. Matwin & T. Muldner (1983) The programming language PROGRAPH: yet another application of graphics. *Proceedings of Graphics Interface'83*, Edomonton, Alberta, pp. 143–145.
29. D. N. Smith (1988) Visual programming in the interface construction. *Proceedings of IEEE Workshop on Visual Languages*, Pittsburgh, PA. IEEE CS Press, Silver Spring, MD, pp. 109–120.
30. F. Ludolph, Y. Y. Chow, D. Ingalls, S. Wallace & J. Doyle (1988) The fabrik programming environment. *Proceedings of IEEE Workshop on Visual Language*, Pittsburgh, PA. IEEE CS Press, Silver Spring, MD, pp. 222–230.
31. T. D. Kimura, J. W. Choi & J. M. Mack (1986) A visual language of keyboard-less programming. Technical report WUCS-86-6, Department of Computer Science, Washington University, St. Louis.
32. M. Najork & E. J. Golin (1990) Enhancing show-and-tell with a polymorphic type system and higher order functions. *Proceedings of IEEE Workshop on Visual Languages*, Skokie, IL.
33. T. D. Kimura (1992) HyperFlow: a visual programming language for pen computers. *Proceedings of IEEE Workshop on Visual Languages*, Seattle, Washington. IEEE CS Press, Silver Spring, MD, pp. 125–132.

34. A. S. Fukunaga, T. D. Kimura & W. Pree (1993) Object-oriented development of a data flow visual language systems. *Proceedings of IEEE Symposium of Visual Languages*. IEEE CS Press, Silver spring, MI, pp. 134–141.
35. T. D. Kimura, A. Apte, S. Sengupta & J. W. Chan (1995) Form/formula: a visual programming paradigm for user-definable user interfaces, *IEEE Computer* **28**, 27–35.
36. G. Karsai (1995) A configurable visual programming environment: a tool for domain-specific programming. *IEEE Computer* **28**, 36–44.
37. E. Glinert (1986) Towards second generation interactive graphical programming environments. *Proceedings of IEEE Workshop on Visual Language*. IEEE CS Press, Silver Spring, MD, pp. 61–70.
38. M. A. Aufaure-Portier (1995) A high level interface language of GIS. *Journal of Visual Languages and Computing* **6**, 167–182.
39. I. Nassi & B. Shneiderman (1973) Flowchart techniques for structured programming. *ACM SIGPLAN Notices* **8**, 12–26.
40. S. K. Chang, G. Polese, R. Thomas & S. Das (1997) A visual language for authorization modeling. *Proceedings of 13th IEEE Symposium of Visual Languages*, Capri Island, Italy, September 1997, pp. 110–118.
41. M. Hirakawa, M. Tanaka & T. Hichikawa (1990) An iconic programming system, HI-VISUAL. *IEEE Transactions on Software Engineering* **16**, 178–184.
42. M. Davis (1993) Media stream: an iconic visual language for video annotation. *Proceedings of IEEE Symposium of Visual Languages*. IEEE CS Press, Silver Spring, MD, pp. 196–201.
43. S. K. Chang, M. J. Tauber, B. Yu & J. S. Yu (1989) A visual language compiler. *IEEE Transactions on Software Engineering* **15**, 506–525.
44. S. K. Chang, G. Polese, S. Orefice & M. Tucci (1994) A methodology and interactive environment for iconic language design. *International Journal of Human Computer Studies* **41**, 683–716.
45. D. Harel (1988) On visual formalisms. *Communications of the ACM* **31**, 514–530.
46. M. W. Maimone, J. D. Tygar & J. M. Wing (1988) Micro Semantics for security. *Proceedings of IEEE Workshop on Visual Languages*, Pittsburgh, PA. IEEE CS Press, Silver Spring, MD, pp. 45–51.
47. P. D. Wellner (1989) StateMaster: a UIMS based on statecharts for prototyping and target implementation. *Proceedings of SIGCHI '89: Human Factors in Computing Systems*, Austin, Texas, pp. 177–182.
48. J. Rumbaugh, I. Jacobson & G. Booch (1998) *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, Addison-Wesley, Reading, MA.
49. Z. Diskin, B. Dadish, F. Piessens & M. Johnson (2000) Universal arrow foundations for visual modeling. *Theory and Application of Diagrams* In: (M. Anderson, P. Cheng & V. Haarslev, eds). Springer, Berlin, pp. 345–360.
50. J. Ebert, A. Winter, P. Dahm, A. Franzke & R. Süttenbach (1996) Graph based modeling and implementation with EER/GRAK. In: *Proceedings of ER'96* (B. Thalheim, ed.). Springer, Berlin, pp. 163–178.
51. M. Minas (2002) Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming, to appear.
52. R. Bardohl, T. Schultzke & G, Taentzer (2001) Visual Language Parsing in GenGEd. *Proceedings of 2nd International Workshop on Graph Transformation and Visual Modeling Techniques GT-VMT'01*, Crete, Greece.
53. R. I. Ferguson, A. Hunter & C. Hardy (2000) MetaBuilder: the diagrammer's diagrammer. In: *Theory and Application of Diagrams* (M. Anderson, P. Cheng, V. Haarslev, eds). Springer, Berlin, pp. 407–421.
54. P. Bottoni, M. F. Costabile & P. Mussio (1999) Specification and dialogue control of visual interaction through visual rewriting systems. *ACM TOPLAS* **21**, 1077–1136.