An analysis of strictly visual codes in programming environments from a semiotic perspective

DELLER J. FERREIRA
email: deller@inf.pud-rio.br

CLARISSE S. DE SOUZA
email: clarisse@inf.puc-rio.br

ROBERTO IERUSALIMSCHY
email: roberto@inf.puc-rio.br

PUC-RioInf.MCC22/96 August, 1996

Abstract: A semiotic analysis of the use of visual programming codes supports the exploration of their limits for the specification of programs. Results suggest why strictly visual representation for program control and modularization structures are inadequate. Nevertheless, visual codes can be combined with textual codes to help programmers to understand and create software in a communicatively rich environment. VEPascal is one such environment being used in the study as a reference for pilot tests. In it, images are based on geometric forms, the composition of which evokes similar concepts to those of the control and modularization structures in structured programming languages. Such images are motivated by perceptions related to semantic aspects of the control and modularization structures represented by them and are modeled by means of L-systems designed so as to be used in processes of understanding, debugging or conversion of software.

Keywords: semiotic engeneering, l-systems and visual programming languages.

Resumo: Uma análise semiótica do uso de códigos visuais de programação é utilizada para explorar seus limites para a especificação de programas. Resultados sugerem porque representações estritamente visuais para estruturas de controle e modularização de programas são inadequadas. Contudo, códigos visuais podem ser combinados com códigos textuais para auxiliar programadores a entender e criar software em um ambiente comunicativamente rico. VEPascal é um desses ambientes, sendo usado neste estudo como referência para testes piloto. Em VePascal, imagens são baseadas em formas geométricas, onde sua composição evoca conceitos similares aos de estruturas de controle e modularização em linguagens de programação estruturadas. Tais imagens são motivadas por aspectos perceptivos relacionados a semantica de tais estruturas , são modeladas por meio de L-Systems, sendo elaboradas para serem usadas em processos de entendimento, depuração ou conversão de software.

Palavras chave: engenharia semiótica, l-sistemas e linguagens visuais de programação.

1. Introduction

Empirical studies Stasko & Lewis (1993), Whitney & Urquhart (1993) reveal that, for a set of algorithms (some mathematical algorithms, sorting algorithms and search algorithms), visual program representations help in the recognition of computatorial constructions. Programming languages that make use of visual representations, not being limited to textual representations of programs, are called visual programming languages. In a visual programming language images are used to represent languages expressions Ambler & Burnett (1989), Shu (1990). In this way programs are shown graphically creating an alternative form ofexpression.The graphic representation of programs, when used to explain their static anddynamic features, helps in understanding the hierarchy and interactions between the program components. However, the real impact of visual representations should be assessed in terms of programming languages in general use.

With the aim of discovering if purely visual representations make an interpreter or compiler interface viable for programming languages, Semiotic Engineering de Souza (1993) is considered in the present study as a tool to assess the expressiveness of visual programming codes. Using this approach, semiotic guidelines are proposed to support the decision making of the designer as to the ways of expressing his intentions concerning the operation and applicability of the system. In this way visual programming environments are seen as messages sent from compiler designers to programmers (Figure 1).

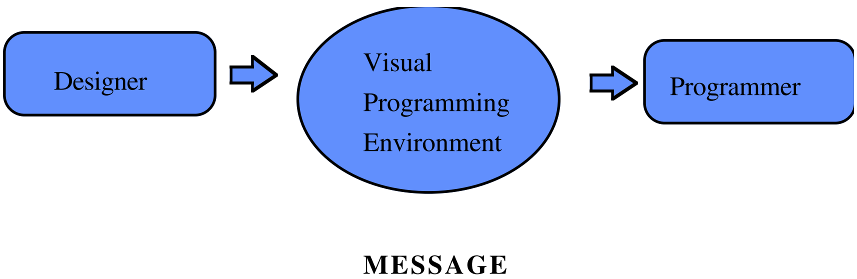

FIGURE 1. Visual Programming Environment seen as a message to the programmer.

In order to evaluate the range of visual representations in the programming process, a visual environment called VEPascal (Visual Environment in Pascal) was elaborated, motivated cognitively and perceptively in terms of the visual signs that represent control and modularization structures in the Pascal language. An analysis according to the principles of Semiotic Engineering of the VEPascal signs is shown to be a better way of fixing a possible generic result which would seem to be semiotically inadequate for representing control and modularization structures in strictly visual codes.

Besides visual codes for representing control and modularization structures, the VEPascal environment also contains textual codes. The integration up of these two codes in a single programming environment seems to offer considerable advantages according to initial interpretability tests of the images used in the language, which indicate a strong perceptive appeal.

VEPascal can be exploited to investigate the learning, codification, debugging and analysis of the code of programs from a visual programming viewpoint. In relation to *learning*, geometric shapes are mapped to the Pascal language control structures, suggesting the flow control over the data.

The emphasis of the visual sentences on the control structures facilitates *codification*, as it leads to a stratified view of the program. In order to avoid introducing additional complexity to the programs, there is a one to one relationship between a subgroup of syntactic rules of Pascal and the visual syntactic rules of VEPascal. The textural and visual codes are generated and shown simultaneously in different windows by employing a textual code generator activated by visual entries during the generation of the visual code. For the *debugging* and *analysis* of the code a Denotation Semantics Meyer (1990) approach is used by animating geometric forms represented by L-systems Lindenmayer (1968). The L-systems dictate systematic variations in the geometric forms and the animation simulates the execution of the program. During the simulation of the execution it is possible to examine the logical structure of the program and the animation makes it possible to follow dynamically the flow of data, enabling one to detect parts never reached, as well as visualizing the paths followed. The programmer can observe the behavior of the program and make judgments about its efficiency, comparing different versions and implementations.

The VEPascal environment is described in Section 2. Section 3 contains a semiotic analysis of the visual representations of VEPascal without loss of generality and its extension to any visual environment which describes control and modularization structures by means of visual codes. Section 4 describes the interpretability tests of the visual representations considered as a reinforcement to the textual representations and the conclusion of the study , following a Semiotic Engineering line, is a tentative explanation to why strictly visual programming codes are inadequate, but may be useful to programmers when they offer, through an association motivated by textual codes, views of general patterns of program structuring and functioning (syntax and semantics, respectively).

2. The VEPascal Environment

VEPascal is a semiotically motivated system which attributes graphic extensions to Pascal language, facilitating the analysis of the generation, testing debugging , and analysis of the complexity of visually described programs. With the objective of providing good visual tools for formulating the mental model of the user, cognitive and semiotic principles were followed in the design of the environment in pursuit of greater simplicity in attributing meanings to the signs of its interface. Aiming at a semiotic formulation, VEPascal was elaborated by stipulating the visual expressions as translations of textual expressions. There is a one to one relationship between the form and the content of VEPascal visual expressions and a subgroup of Pascal expressions. Despite cognitive and perceptive motivation at the language *token* level and of isomorphism at the grammatical level, in VEPascal the visual code does not play the main role, but is only used to provide an alternative option of expression. The textual and visual codes are presented at the same time in different windows and mutually reinforce each other within a supplementary framework of expression.

The environment is formed by a code generator and a simulator for executing programs (see Figure 2), where the images are presented as an informative supplement to the text of the program. The visual representations together with multiple viewing help programmers to visualize the structure of the program and to perceive its modules at each level of abstraction. The idea of multiple views is based exactly on structured programming and the Pascal language is used to describe the programs text.

The reason for developing a language based on Pascal is due to the fact that this is a structured language and representative of the pattern of traditional imperative languages. Pascal covers the majority of the basic constructions of procedural languages, being a type of *lingua franca* in computer science Nievergett & Hinrichs (1993).

In VEPascal the program is generated visually and in a structured way by a graphic interface, prompting visual representations of the control and modularizations structures. The top-down approach inspired the 2_D interface used in constructing the program to provide a feedback of its hierarchy. After activating the program symbol, an area for the visual code is opened up at the main program level where the visual representations are positioned after being selected. Each representation in the visual code area can be chosen to activate a new code area at the selected control structure level. The structured approach allied to the use of a 2 _ D interface is useful for presenting the visual code in a clear and stratified way, enhancing the expressive powers of the environment. The static code is visualized level by level by means of a navigator which allows top-down and bottom-up visualizations of the program. As can be seen in Figure 2, at the program level, the visual code of two "For" structures is shown. Should the user want to visualize the part of the program containing the "If-Then-Else" structure, the second representation of the "For" structure (circumference on the right in the visual code window) should be selected.

There are three windows in generating the visual and textual code. Two textual code windows and one visual code window (VEPascal, Pascal code and a third window whose title varies according to the level). The VEPascal visual code window is an Input-(I) window, the Pascal code window is an Output-(O) window and the third window is Input/Output-(I/O) one. The Output window is global, showing the full text of the program. The other text window contains the text of the program modules, being a reference of each program level. There is always consistency between the three windows; if a change alters the contents of one of the windows, the contents of the other windows are immediately updated. Generation of the code starts after selecting the program symbol. By direct manipulation, the user selects the control and modularization structures pressing on icons in the interface.

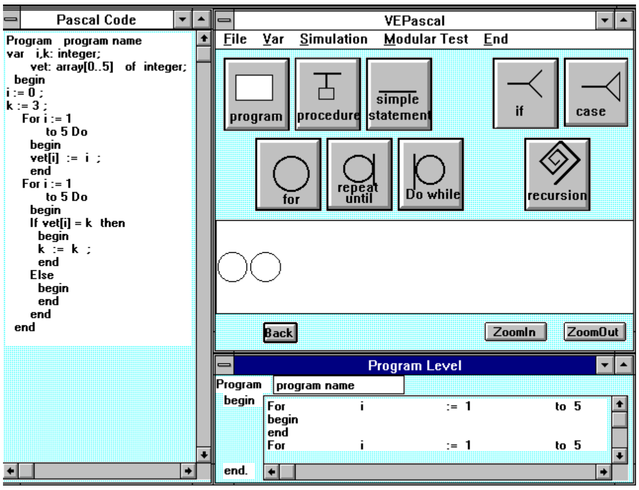

FIGURE 2. Environment for generating the visual and static textual codes of the program

The composition of the program includes textual entry of the variables, data structures, simple commands and control conditions. During the selection of the control structures their syntactic representations described by L-systems are associated . L-systems are rewriting systems characterized by the parallel application of their rules. They lend themselves easily to a geometric interpretation, Turtle Geometry Prusinkiewicz (1986) being the most famous. Turtle Geometry Abelson & diSessa (1980) is a mathematical system, where geometric forms are constructed by moving an imaginary turtle. The turtle's movements are established by changes in the state of the turtle. This state is a triplet (X, Y, α), in which the Cartesian coordinates (X, Y) give us the position of the turtle and the angle α supplies the direction of the turtle's head. Figure 3 exemplifies a sequence of geometric commands of the turtle.

The interpretation of the turtle takes advantage of a basic set of symbols of the L-system alphabet, which are:

- F moves forward making a line segment. The state of the turtle alters to (x',y', α), where $x' = x + d\cos\alpha$, $y' = y + d\,\mathrm{sen}\alpha$, d a pre-fixed distance. A line segment is made between the points (x,y) and (x', y').
- f moves forward without making a line.
- + turns counter-clockwise according to a given angle δ The next state of the turtle is (x, y, μ $+\delta$)
- turns clockwise according to a given angle δ The next state of the turtle is (x, y, α. - δ).
-) angle δ increment by a given value
- (angle δ decrement by a given value.
- [pushes the current state on a stack of states.
-] pops the top of a stack of states and updates the current state.

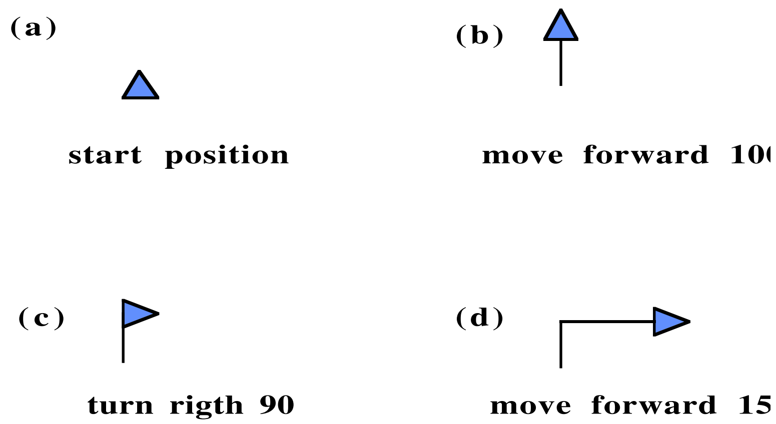

FIGURE 3. Geometric commands which dictate the turtle's movements.

In VEPascal, the L-systems describe geometric forms which represent the control over the flow of data. Such geometric forms are created according to cognitive aspects. The capacity of perception and generalization are important operations in cognitive processes. Generalizations may be obtained through recognizing regular patterns which are used in evaluating new facts. Perspective motivation, and the capturing of regularities, therefore facilitate the attribution of meanings to the visual language expressions. VEPascal exploits images based on perceptive and on analogies as to the potential meanings of the perceptive structures to aid programmers to form ideas based on observing images or forming images based on mental models. Geometric variations of the forms describe regularities used as an aid for one to infer the dynamic behavior of the program.

The simulation of the execution is implemented by animating the structures so as to provide visual means of support for the testing, debugging and analysis of the efficiency of the program (Figure 4).

L-systems are fundamental in animating VEPascal, being a formal representation which allows elegant animation algorithms through Turtle Geometry. The interpretation is made through a recursive algorithm which applies the rules of the L-systems and the geometric interpretations when an interpretable symbol is found.

The animation of an algorithm is a dynamic graphic illustration of the algorithm, with the function of making the dynamic properties of the program explicit. It is used to inspect the dynamic properties during execution. The feedback provided by the simulation can be used in various situations. For example, for instructional use in teaching " Basic Algorithms." In the case of programmers the graphic simulation of programs makes it easier to detect possible errors, redo the program and analyze its efficiency. In building small programs its is common to develop first a simple version of algorithms to be used in a subsequent analysis of the efficiency. The analysis of the efficiency seeks to optimize the version of the algorithms and its implementation. Certain steps to be carried out are to identify the slower processing parts and check what are the most frequently executed instructions. VEPascal describes by means of animated images the control over the data and feedback from the execution time providing modes of comparison between different versions of algorithms and different implementations.

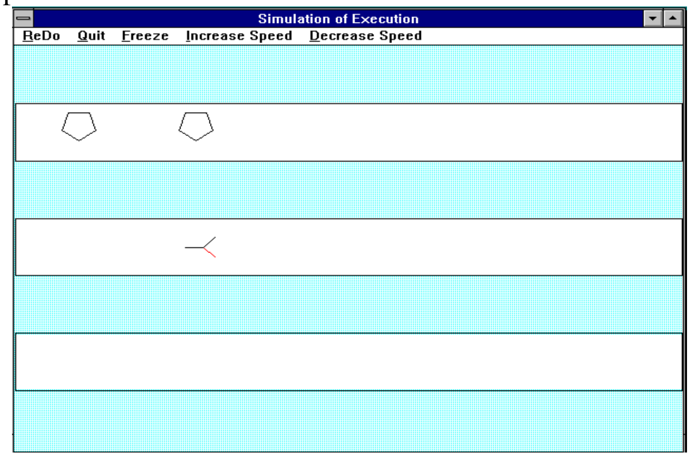

FIGURE 4. Simulation of the "Sequential Search" program

In VEPascal the monitoring. of the program is done through animation level by level. The animation presents the hierarchy of the program in a structured form passing through different levels of abstraction. The speed of animation of each structure reflects its computational complexity, the greater the complexity, the greater the speed. The animation of algorithms is a factor which helps to make VEPascal more expressive by showing the dynamics of the program.

During animation the following tasks are possible:

- freeze animation
- control over-all speed of animation
- examine the execution of parts of the program
- compare different versions and implementations of the algorithm
- visualize the data path
- evaluate the efficiency of the algorithm

3 - Analysis of the adequacy of visual representations of control and modularization structures

Programming languages are typically implemented to be interpreted by computers, although they should be intelligible to programmers Soloway (1986). In analyzing the degree of interpretability of strictly visual representations of control and modularization structures from the point of view of the programmer, adequacy will be assessed from a semiotic viewpoint. To provide a better presentation of the analysis, we will take a look initially at the analysis of VEPascal visual representations described in Section 2. In this way, the concepts will be discussed in a less abstract context such as motivation so that the result may be extended to any strictly visual code of control and modularization structures of programs. The analysis is based on concepts of Semiotic Engineering, which establishes guidelines for forming interface languages. In terms of adequacy, there is a subtle difference between a visual programming language and a visual interface language. Visual interface languages typically use tokens which bring into play the expressive potential of the contextualized instances of the fields of action of the applications such as, for example, a sheet of paper for word processors. Visual programming languages are program builders and have tokens therefore which represent abstract programming concepts, giving visual representation to what is abstract by nature. This difference should not prevent a visual programming language from following the four guidelines of Semiotic Engineering for interface languages, since they are sufficiently generic.

Based on Umberto Eco's (1976) four parameters proposed in the theory of sign production, de Souza (1993) proposed a set of four guidelines for building the interface language. They can be paraphrased as:

(A) *Interface designers* should use interface signs that are recognized as codified expressions of the desired contents.

(B) Expressions should be chosen that are recognized as an instance (particular case) of a *well-established and preferential type* of expression system.

(C) The *interface* signs related to the computational modeling of solutions should be chosen from within the *universe of signs of the application's field of action and not from the universe of compuer signs proper*, composed files, folders, disks, windows, programs and other items.

(D) Expressions derived from a *rule-based system* should be used.

Programming languages have static and dynamic features and can be taken as an interface instance between a programmer and a program generator application. Our semiotic analysis of VEPascal control representations will be carried out in two stages: the first taking into account the static code, and the second, the dynamic behavior of the program.

In terms of the static code, in order to be in accordance with the first guideline, the visual signs which statically represent the control structures should be pre-existent. Nevertheless, such signs in VEPascal have been invented, bearing in mind perceptive and cognitive aspects. The control structure signs are forms which visually evoke their behavior, that is to say, forms which represent the control over the flow of data. Figure 5 shows the conditional sign "If-Then-Else", where the notion of two paths to be followed by the data is transmitted by a bifurcation. The recursion notion is also shown, by means of a figure similar to a spiral, where each straight segment represents a recursive call. Thus although the signs are perceptively and cognitively motivated, guideline A cannot be satisfied, since the contents of concepts related to programming language control structures do not have a generally agreed upon physical or graphic substance.

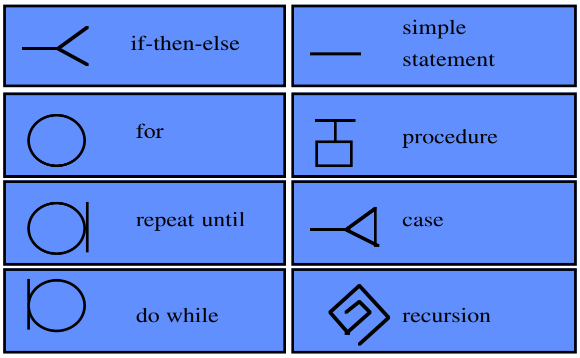

FIGURE 5. Association of language structures to geometric forms

Guideline (B) stipulates that the chosen signs should be instances of a *preferential* chosen type for the contents of what one wants to represent. The preferential expressive type for the meanings of control structures in programming languages is **textual** owing to the abundance of programming languages in which these structures appear textually, and the total scarcity of generic visual languages that have visual representations of such structures. As in VEPascal the control structures are visually represented, consequently the second guideline is not met. Figure 5 also illustrates the "Repeat Until" and "Do While" representation which are meant to mean test after the interactive processing and test before the interactive processing, respectively. In both images, the control is returned to the initial point of processing giving the loop notion, which is shown as a circumference.

What one finds in fact, as far as guidelines (A) and (B) are concerned, is that from the moment when all the simple (lexical) signs of language are invented, guideline (B) is not observed consequently affecet and, as it cannot have as a preferential type something that demands that the tokens (instances) of the expressive system be invented.

The representation of VEPascal control structures is also not in accordance with the third guideline, as the signs are not taken from an expressive universe pertaining to the programming sphere, but to an abstract sphere - geometry. Geometric forms suggest the behavior of the dataflow. For example, the rectangle is the conventional representation of the program. The rectangle was inspired by the form of a box, seeking to express an entity which encompasses the control and modularization structures. The simple command is shown as a dash, signifying a portion of a sequential flow. A procedure call is shown by a rectangle hanging in a straight segment, signifying a deviation to an entity which, like the program, involves other structures.

Finally, Guideline (D) is not obeyed in building the geometric forms, where there is no need for a description by means of rules, as the static representation is immutable and unique.

We have now evaluated the adequacy of VEPascal visual representations in relation to their static form in order to discover the degree of interaction of the visual signs with the programmer, and verified that all the guidelines for designing good interface language were some how discarded in relation to the visual signs. It can be said, therefore, that a semiotic analysis of the VEPascal static visual signs is unfavorable. However, the analysis so far has only explored the static code of the program. The analyses of guidelines (A) and (B) can be extended to the dynamic code whilst those of guidelines (B) and (D) need to be revised. The analysis of the representation of the dynamic features of the program requires a revision of the use of L-systems in VEPascal and the introduction of the concept of Denotational Semantics.

Denotational Semantics is used to express effects of the program execution. The denotational approach covers an area of meanings, which is a set of abstract entities which represent the meanings of all the language constants, and a mapping of the relationships of the language in the relationship in the area of meanings. There is a scheme of translation associating a meaning (denotation) to each language program (Figure 6).

The denotational description of a programming language is given by a set of M meaning functions associated to a set of grammatical constructions. Each function is of the M : T --> D form, where T is a construction such as, for example, an instruction or a program.

In VEPascal the control structures which are visual descriptions of Pascal control structures have a set of rules associated to them. When the number of applications of the rules varies, a new image is created. The semantics of the control structures are mapped by L-systems, providing efficient and elegant animation algorithms.

The animation suggests the flow of data using geometric regularities governed by the L-systems. For example the FOR structure is denoted by:

Mfor : FOR --> FOR L-system

ForL-system

Axiom: A

Rules: A --> F + A

Rotation Angle δ: 360º /n, with n being the number of applications of the rule

Stop: n

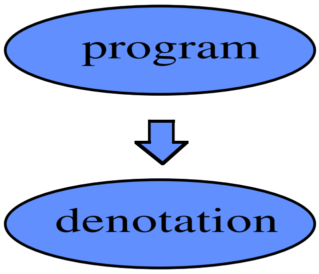

Figure 6 Translation of a program for a meaning

Size of the dash: the size of the dash is proportional to the rotation angle being equal to cos ((180º - δ) /2)* T, with T a variable controlled by the dimensions of the area of the drawing.

Thus, a FOR structure repeated four times is represented by a square, repeated six times by a hexagon and repeated many times by a circumference. The obvious motivation for this sign is the association with the idea of “going round in circles.” From the formal viewpoint, therefore, the circumference is the “upper limit” of an n-sided polygon, with n tending to infinity. The lower polygonal limit, however, is 3, since one cannot draw polygons with less than 3 sides..

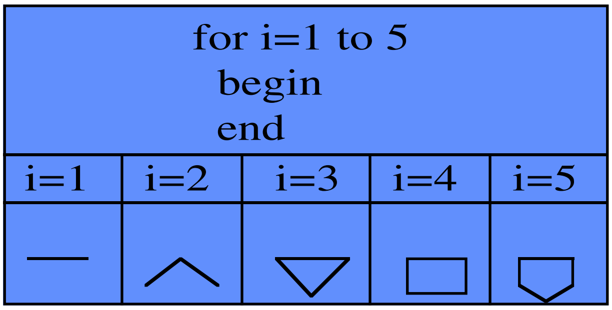

FIGURE 7 Animation of the FOR structure with five repetitions

In the same way as for the FOR structure, concepts of Denotational Semantics were followed in representing the other VEPascal control structure, with a translation of Pascal control structures into L-systems, which are later interpreted by Turtle Geometry. In this way, one can say that guideline (B) is partially fulfilled. Figures are used in place of text, but a well-known technique, Denotational Semantics, is followed in attributing meanings to the language constructions. Guideline (D) is also obeyed, as the construction of the geometric forms which represent the behavior of the control structures is carried out by means of L-systems, which codify a system of rules and capitalize on motivational data to introduce regularities and interpretative motivations through animations.

In relation to the representation of dynamic behavior, the VEPascal visual codes satisfy guidelines (C) and (D), with better results than the static representation, but without there being a significant difference in adequacy.

The analysis of the signs for representing VEPascal control structures in their static and dynamic aspects does not obtain such favorable results as one would like. However, the reader might well ask if the analysis is unfavorable simply because the visual representations were badly designed, which raises, therefore, a second question - would the result of the analysis have been better if we had considered other visual representations for control and modularization structures. To answer this question we have to go back to the four guidelines of Semiotic Engineering.

The first guideline requires the existence of visual expressions, which are well-codified and recognizable as such. Despite there being different visual programming environments such as the PICT Glinert and Tanimoto (1984), which take advantage of different visual representations, there are no standardized visual representations for control and modularization structures. Therefore, as far as both their static and dynamic features are concerned, this guideline will never be complied with no matter what the visual modeling of the control and modularization structure of the Pascal language may be.

The second guideline establishes that the type of expression used in the language should be well-established. The first programming languages and the most widely used current languages are textual. This fact allied to the non-existence of visual representations which are recognized as codification's of control and modularization structures has led to a TEXTUAL-type of expression being preferred for programming languages. Accordingly, it is not possible to satisfy the second guideline by means of VISUAL-type representations.

The third guideline says that the I/O elements should be selected from within the programming universe, again falling back on the argument of the second guideline, as there are no visual programming entities used to represent the static or dynamic aspects of control and modularizations structures, which makes it impossible therefore to satisfy this condition.

The fourth guideline requires that expressions derived from a rule-based system be used. As static representations are immutable, there is no reason for their representation to be described by means of rules. While for dynamic representations it is possible to elaborate rules as shown in VEPpascal.

After discussing the four guidelines of Semiotic Engineering, we can see that creating visual representations of the programs is a semiotically difficult process, as it is not a usual form of representation. We can put forward the most generic result that strictly visual codes for representing control and modularization structures in imperative programming languages are semiotically inadequate.

4. Conclusions based on pilot interpretation tests.

The purpose of the tests reported in the following is to put forward the use of semantically elaborated visual codes as a supplementary form of visualization. The five pilot tests were carried out in a class of 10 graduate students with a 100% success rate. Such tests consisted of the following:

4.1. PERCEPTION OF IMAGES TEST

To associate control structures and their visual representations. This test was done to verify if the selected images are perceptively motivated in their static form, within the universe of selected geometric forms.

4.2. RECOGNITION OF CODE PATTERNS TEST

To identify parts of programs during the simulation of the execution. .Given the program which initializes an array and which based on a value to be found implements the “Sequential Search” algorithms as shown in Figure 2. The program code was presented and asked to show visually the animation of parts of the code, with the aim of seeing if the visual representations provide a local view of the program.

4.3. GLOBAL FEEDBACK OF THE PROGRAM BASED ON TEXT TEST

Given the textual code of a program in Pascal and different animation’s, test participants should correctly associate which animation refers to the text in question. For example, given the code of a program which implements “Binary Research”, the entry of data and the key, the students were able to formulate the execution of the program mentally and make comparisons with the animation of programs being shown.

4.4. GLOBAL FEEDBACK OF THE PROGRAM BASED ON ANIMATION TEST

To interpret the execution through observing the animation of the geometric forms. Given the simulation of a program, the subjects observe its dynamic behavior inferring its logic. For example, the FOR structure is inferred from the repetition of the forms which compose the internal code of its *begin* and *end* and the growing number of sides of a regular polygon. One could detect the simulation of a recursive procedure with 7 recursive calls. And through the change of color in part of the figure of the IF structure one could identify the direction of the next IF (THEN or ELSE). This test detects if the representation of the dynamic code provides additional information to the programmer.

4.5. FEEDBACK OF THE PROGRAM HIERARCHY

The subjects should answer questions about the depth and breadth of the program to perceive the association of each band of the animation window to each nesting depth of the structures, associating the simulation of the program to a hierarchic reading of it. This test indicates if the visual code used is a good way of showing the program hierarchy.

4.6. CONCLUSIONS

All the above tests had a 100% success rate in the small group of subjects that took part in the experiments. Tests on a larger scale would be indispensable for an empirical validation of analytical results proposed here. However, we used this small-scale experiment as a measuring rod for assessing the general lines of the thought followed by this research effort.

As for Computational Semiotics we found that strictly visual codes were inadequate for transmitting control over the flow of data in programs, considering the guidelines established by de Souza (1993). This semiotic analysis reveals an important problem of adequacy between the type of preferential expression for programming languages, **textual**, and the type of expression in the field of visual programming languages, **visual**. Generally agreed on visual codes are specific (for example, traffic codes and the XEROX Star desktop code), and their generalization for programming languages is very difficult. However, the VEPascal environment as a whole, does not seem to be inadequate, as it uses the **text** as its preferential type of expression. Visual comprehension of the described programs is favored principally by the use of visual expressions that are translations (one to onemapping) of a set of expressions from the Pascal language and by the explicit use of text. The visual expressions describe the positioning of the virtual representations of control structures in a 2_D interface. In this way the programmer naturally perceives the program schema by means of a stratified view During the production of signs, where there is no available convention in the culture, we have to invent them. In VEPascal the signs are invented based on the perception of geometric forms. The pilot tests elaborated to assess the quality of the representations and visual expressions taking into account cognitive and perceptual aspects attest to promising potential choices in this field. Testing the adequacy of VEPascal according to a cognitive and perceptive perspective , we find the use of regular patterns in the animation of geometric forms and images with a perceptive appeal, providing a rapid association of the images with their meanings. The programmer easily deduces the functionality of each control structure. The perceptive motivation provides a static feedback of the control over the data as, for example, by using the circle to represent the "FOR"structure, reflecting the loop notion. Animation based on rules makes it possible to observe the dynamic behavior of each structure. The "FOR" structure is animated by regular polygons which tend to be transformed into the circle so that the number of repetitions is explained by the number of sides, indicating in which *loop* the program is to be found. With each repetition a regular polygon is displayed with the number of sides indicating how many times the condition was tested. In this way users can easily recognize program patterns. The tests also show that the hierarchical organization of the program is easily noted and that the users are able to recognize a program by examining the simulation of the execution.

The next step in the research is to elaborate and run more profound tests concerning VEPascal, as well as to develop and extend the semiotic analysis to the language features, including those not explored in the initial analysis described in item 3. We plan to give the same treatment to other programming environments, seeking to obtain generic results in the sphere of visual programming languages.

This work was developed from my PhD. studies with Clarisse S. De Souza and Roberto Ierusalimschy at PUC-Rio.

References

ABELSON, H. & DISESSA, A. A. (1980). Turtle geometry. *MIT Press Series.*

AMBLER, A. L. & BURNETT, M. M. (1989) Influence of visual thecnology on the evolution of language environments. *IEEE* . October. pp. 9-22

CARD, S. K. ROBERTON, G. G .& MACKINLEY, J. D. (1991). The information visualizer. *Proceedings of CHI'91*, ACM Press. pp 181-188.

DE SOUZA, C. S. (1993). The semiotic engineering of user interface languages. *International Journal of Man-Machine Studies.* N. 39. pp. 753-773.

ECO, H.(1976). A theory of semiotics. Bloomington, In.*Indiana University Press.*

LINDENMAYER, A. (1968). Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*. Vol.18. pp. 280-315.

MEYER, B. (1990). Introduction to the theory of programming languages. *Prentice Hall.*

NIEVERGELT, J. & HINRICHS, K. H. (1993). Algoritms and data structures *Prentice Hall.*

PRUSINKIEWICZ, P. (1986). Graphical applications of l-systems. *Proceeding of Graphical Interface'86-Vision Interface.* pp. 247-253..

SHU, N. C. (1988). Visual programming. *Van Nostrand.*

SOLOWAY, E. M. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, Vol. 29. N. 9, september. pp. 850-858.

STASKO, A., BADRE, A. & LEWIS, C. (1993). Do algorithm animations assist learning? an empirical study and analysis. *Interchi*. pp. 61-66.

WHITNEY, R. E. & URQUHART, N. S. (1993). Microcomputers in the mathematical sciences:

Effects of courses, students, and structors. *Academmic Computing*, Vol. 4 N. 6. pp. 14-
18, 49-53.