STRUCTURED DEVELOPMENT OF GRAPH-GRAMMARS FOR ICON MANIPULATION

José Javier DOLADO
Facultad de Informática, 20.080 - San Sebastián, Spain.

ABSTRACT. In this work we are showing a structured process to build a grammar for icon manipulation. We presuppose that the object to be manipulated in the computer screen can be stated as a set of relations among its parts. We describe a procedure to generate a program that manipulates the object, guaranteeing that only objects with those properties will be constructed, and that every instance of that object is allowable. The formation rules for the object are stored in terms of attributed graph-grammars productions.

Keywords: programmed graph-grammars, visual editors development, icon manipulation, reusable software engineering techniques.

I. INTRODUCTION

The use of syntax-directed editors for visual programming is well-known, and with that purpose have been applied in [1], [10] and [13]. We are interested in developing these editors in a systematic way for a specific area. In that respect we state the given work in [2], where they show a process to automatically generate visual syntax-directed editors. The example and main thrust they give is for a conventional programming language, although noting that the applied area also includes the structured and unstructured graphs. In this way our attention is explicitly directed according to N.C. Shu [15, p.12] to the "diagrammatic and iconic systems", and according to B.A. Myers [14, p.110] to "flowcharts, graphs and graph derivatives" specification techniques.

The other basis for our approach was founded in [11], [12]. There the use of attributed programmed graph-grammars -APGG- is proposed as an environment to develop syntax-directed editors for graphics [12], being in this case the analyst who develops the grammar, based on his or her experiences and other examples of grammars. The purpose of this paper is precisely to aid the development of any grammar.

The structured approach we propose here for that development consists of a translation to a programmed graph-grammar the whole set of relations that define an object. Afterwards, the grammar only allows to build correct objects. Moreover, at each step it can be known the allowed actions to complete the defined object. Some previous works of the author have shown the feasibility of this approach with a specific type of symbolic methodology [8], [9].

The first step of the method is to define the properties of a specific diagram. Second, and associated to the previous diagram, a graph-grammar and a control program are designed. This set is configurated in such a way that it allows the basic manipulation operations of a diagram (addition and deletion).

In Section II we make some observations about the diagrams we deal with and how to define them. In Section III we present the APGG and the necessary process to construct a grammar that manipulates the diagram defined through the procedure expressed in paragraph II.1. In Section IV we give an example of the use of this method.

II. DESCRIPTION OF A DIAGRAM

The main interest in a diagram is how the set of symbols are organized in it. Given a diagram, the relations that concern our work are the adjacency relations, more specifically the adjacency between two elements, which will be expressed graphically by connectedness and proximity. Thus, we will say that if two elements do not show graphically their relation, there is no relation between them.

This relation -$\mathbb{R}$- is defined as: $a\mathbb{R}b$ implies that the elements *a* and *b* exhibit on the computer screen some graphic characteristics (connectedness, etc.) that show clearly *a* and *b* have some type of relation -at least, to the specialist-. Therefore, we distinguish between: *a)* what the relations among elements are; *b)* how those relations are shown on the computer screen (usually approaching and connecting symbols). What is relevant to our work is the point *a)*, because part *b)* depends on the efficiency of graphical routines of a specific computer and the specific topic of the symbols of the diagram.

Our aim is that a program can create and compound diagrams correctly, independently of what is their graphical expression and relative position of

the objects on the screen. Other routines will do this properly. The basic scheme which allows the interaction with the software in our framework is the "direct manipulation" approach, assisting the user with appropriate messages in the application of the rules.

The simplest operations to be considered are

- addition of a symbol, a new symbol is attached to the host-graph. Only one relation $\mathbb{R}$ is manifested at this time.
- addition of another relation $\mathbb{R}$ between symbols already existent (to connect symbols or to approach them).

The reciprocal operations are

- deletion of a relation. It implies the deletion of a coupling (and maybe graphical rearrangement).
- deletion of a symbol. It implies the deletion of a node and every arc connected to it.

The following paragraph is devoted to make concrete these operations over a procedure so that be as general as possible. In Figure 1 the operations are expressed in the form of graph-grammars productions [5], [12]. Addition and deletion operations previously defined are supported by the transformations of Figure 1. So we should adapt the entity's definition to those classes of operations, obtaining a set of productions in which every component is defined.

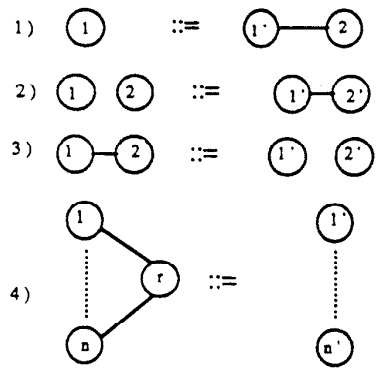

Figure 1. Four classes of operations: 1) addition of node 2 to node 1; 2) addition of a relation between 1 and 2; 3) deletion of a relation between 1 and 2; 4) deletion of node r

II.1. Definition of Graphical Entities.

The entities will be defined describing their parts. Amongst the parts, so far, only relations in number and type are stated. The basic procedure for defining graphical entities is as follows

Procedure DEFINE
begin

1) define the list of different components -symbols-
2) for each component -symbol- define its possible adjacents
3) define for each symbol the maximum and minimum number of adjacents
4) define forbidden combinations of symbols

end.

An attributed graph will support this information. The task is to design a grammar with a set of productions in the form of Figure 1, in such a way that we can build whichever diagram defined through Procedure DEFINE (see III.2).

III. PROGRAMMED GRAPH-GRAMMARS FOR ICON MANIPULATION

III.1. Programmed graph-grammars.

An attributed graph-grammar is a grammar such that its formation rules are defined in terms of graphs [6], [7]. Such rules, named *productions*, are composed of five components.

- two graphs, the *left-hand* and *right-hand* side of a production (it is a straightforward extension of the string-grammars case).
- the *embedding transformation* takes into account the peculiarities arising from subgraph replacement. In our case, this embedding transformation is trivial because the introduction of new elements does not mess up the original graph.
- by means of the *applicability predicate* certain conditions to fulfill by the left-hand side of a production in order to be replaced by the right-hand side can be formulated. This enables us to express constraints on the attributes.
- a finite set of partial functions: the node *attribute transfer functions*.

The *direct derivation* of a graph g' from a graph g by means of a production p is defined by the following procedure [5]

1) Check whether the left-hand side of the production occurs as subgraph in g and check whether the applicability predicate is "True" for this occurrence. If both conditions are fulfilled go to step 2).

2) Replace the left-hand by the right-hand side.

3) Attach attributes to the inserted right-hand side according to the set of functions.

A *programmed graph-grammar* specifies in a control program (or control diagram) (Fig. 2) how the rules (productions) should be organized in order to derive a specific result. The entities delineated are the set of all attributed-graphs which can be derived in the following way:

1) Start with an initial graph.

2) Apply productions in an order defined by the control program. After successful application of a production, a Y-edge (yes) in the control diagram is tracked, while the tracking of a N-edge (no) is caused by the failure of a production. Adding explicit control by programming, we also choose simple embedding transformations for the underlying graphs rewriting steps.

3) Stop the derivation sequence when the final node Z in the control program has been reached.

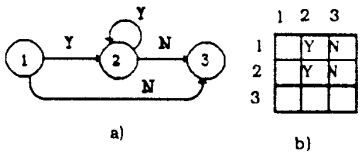

Figure 2. Matrix b) corresponding to the control program a)

In a control program we state the grammar which constructs our desired graphs. The control program defines the permitted rules at each step. Examples of programmed graph-grammars can be found in [5], [6] and [7].

III.2. Designing the grammar for our specific task.

The set of productions is obtained defining -in the general case- $2n+2$ productions, being n the number of different graphic symbols (n for addition of a node, n for connecting two nodes in the graph, *1* for symbol deletion and *1* for connection deletion). The applicability predicates are built depending on the relations among elements and the cardinality of the adjacencies; the embedding transformation is trivial and the attribute transfer functions depend on the attributes. The following extension of an APGG comes from the specific considerations done about the two attributes considered here, number of adjacents and their types:

a) number of adjacents: the maximum number is a property that can be expressed in the applicability predicate of a production. To ensure the minimum number of adjacents we are obliged to apply productions, and the diagram is not valid until such characteristic is verified in the corresponding predicate (see procedure PUSH-TEST below).

b) type of nodes: since the Π_i are referred -in part- to the type of node, given the failure of a production we can associate the applicability of the next production to every symbol. This will imply the definition of table TB associating productions to symbols.

So, once the set of productions is obtained, two matrices remain to be constructed: *a)* TB defines for every element the productions that can be used to modify their attributes; *b)* CP is the control program to order the productions. Let P be a finite set of productions and ST a stack. A *control program* -CP- over P is a graph with the set $P \cup \{O, Z, PT\}$ as node labels and the set {Y, N} as edge labels. Furthermore, the following conditions hold true.

- O and Z are the initial and final node
- P is the set of productions
- ST is a stack which contains graph nodes
- PT is an action PUSH-TEST

PUSH: 'push v', being v the set of nodes involved in the last production P_i applied to the graph. When dealing with node deletions the instruction is "push v except the node deleted". After apply TEST.

TEST is a procedure which consists of

a) stack -ST- condition: empty or not.

b) test of nodes in the stack. The test consists of testing if the node attribute has fulfilled its lower limit.

c) application of a production depending on the node n. This is accomplished by a matrix TB.

TB contains the productions which are to be applied when the test is true for a node n. After the stack has been treated, we can continue in every production. The set v is pushed to the stack in order to check if some axiom of v has been violated, thereby making a requirement to apply productions to restore the node.

Let us call EAPGG *-Extended APGG-* an APGG with the table TB, stack ST and with the PUSH-TEST procedure applicable to the nodes of the CP.

The purpose of defining the stack ST and the table TB is to consider the problem that appears when programming the numerical lower limit of a node. A node is popped out the stack when its properties have been restored.

The CP must be designed with these characteristics:

- allowing the application of $P_1, \ldots, P_n$, Z after O, $P_1, \ldots, P_n$;
- applying productions to every node until they fulfill every characteristic set forth in the description. So, we must apply PUSH-TEST after each production that is able to alter a node.
- In order to empty out the stack ST apply PUSH-TEST just before reaching Z. The schemes are shown in Figure 3 and the correctness of these considerations are explained next.

III.3. Correctness

Next, we demonstrate that these last conditions are necessary and sufficient to construct every possible graph and only that kind of graph (those graphs that are defined through Procedure DEFINE).

Thus, we are looking for the way in which every characteristic defined in Proc. DEFINE is stated in a EAPGG. More formally, we want to construct an EAPGG in such a way that the language generated is the set of graphs of Proc. DEFINE. The next proposition establishes the skeleton of a grammar G generating the language L in such a way that L is the set of attributed graphs fulfilling the properties of Proc. DEFINE. Let us call DL that language and GDL its grammar. This proposition is similar, but more general than, that of [7], except that here we impose conditions over a stack ST.

Proposition. Given an EAPGG with the CP verifying

a) every production can be used at any node

b) every graph generated by CP fulfills Procedure DEFINE

c) The stack ST of CP is empty when node Z is reached

then this EAPGG is a GDL, i.e., generates the language DL.

Proof. An EAPGG is a GDL if it is able to construct every DL graph, and if every graph that can be constructed is an DL. This is true because

- if the stack ST is empty then every node of the diagram has been treated
- if every diagram fulfills the properties stated through Proc. DEFINE, it is a correct diagram
- if every production can be used at any node we are able to construct every desired diagram.

So, the task thereafter is to transfer to a control program CP the structures collecting the characteristics *a), b)* and *c)*. Part *a)* is achieved by allowing the application of P_1,....., P_n, Z after O, P_1,....., P_n; *b)* is fulfilled by applying productions to every node until they fulfill Proc. DEFINE. So, we must apply PUSH-TEST after each production that is able to alter a node. In order to empty out the stack ST (condition *c)*) apply PUSH-TEST just before reaching Z. The schemes are shown in Figure 3. In [7] the previous productions used were taken into account, and that constrained the next productions to apply. In order to generalize the ideas, the stack contains nodes instead of productions.

At this moment it only remains to define the set of productions to apply for each type of node. The matrix TB is constructed in the following manner (its skeleton is given in Figure 3 e)):

a) to select among the list of symbols, those which have a lower limit of adjacents distinct of 0 (zero).

b) to select among the set of productions, those which alter that limit, including the node deletion production.

When the set of productions and CP have been built, it only remains to define the interface and to flow through the steps that follow: *a)* start with node O; *b)* apply productions following the indications of the CP until node Z is reached.

IV. EXAMPLE

In this section we are going to demonstrate how to proceed with this method when a programmer needs a software for icon manipulation. The example is a simple diagram, but containing every characteristic described.

Given four geometric figures: Circle -C-, rectangLe -L-, rhombUs -U-, Arrow -A- with the following

- adjacencies: C-A, C-L, A-C, L-C, L-U, L-L, U-L,
- with the following limit in the number of adjacents *-adj-*: $2 < adj(C) < 4$; $1 < adj(A) < 3$; $0 < adj(L) < \infty$; $< adj(U) < \infty$.

(meaning, for example, that a symbol C (circle) can not have less than 2 and no more than 3),

- with the following forbidden interactions: rectangle-rhombus

we will obtain a program for manipulating those icons through 1) diagram definition, 2) rules of the grammar definition, and 3) control program and TB definition.

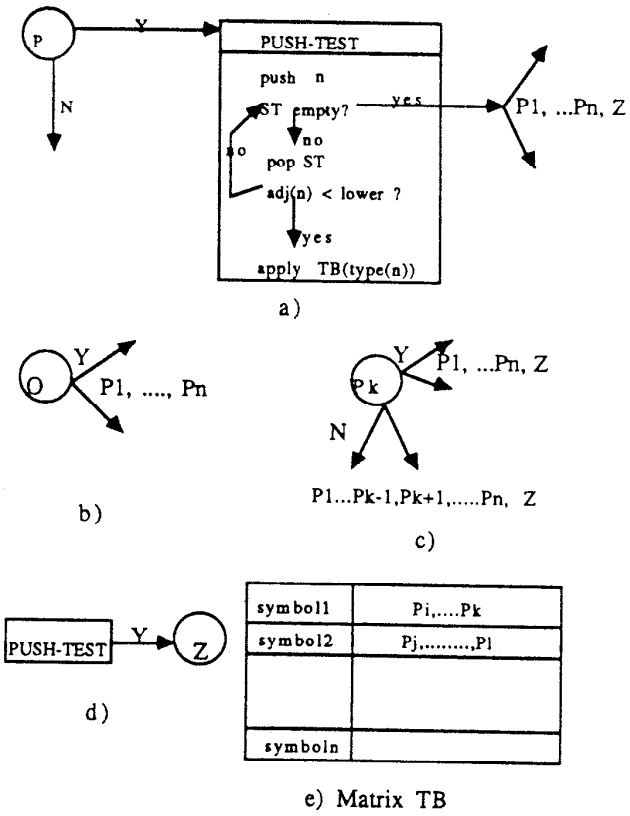

Figure 3. Subschemas of the control program: a) PUSH-TEST; b) initial node; c) intermediate node of the CP; d) last node of the CP; e) table TB

IV.1. Diagram definition. (Procedure DEFINE)

Step 1: Define the list of symbols in question: {C, U, A, L}

Step 2: Definition of relations -ℝ- among them

$C \mathbb{R} A, C \mathbb{R} L, A \mathbb{R} C, L \mathbb{R} C, L \mathbb{R} U,$
$L \mathbb{R} L, U \mathbb{R} L.$

Step 3: Definition of the number of adjacents -adj-.

For all node *n* of type "circle", $n \in C$, $2 < adj(n) < 4$;
similarly $n \in A$, $1 < adj(n) < 3$;
$n \in L$, $0 < adj(n) < \infty$;
$n \in U$, $0 < adj(n) < \infty$;

Step 4: Definition of forbidden combinations:

$L \mathbb{R} U; U \mathbb{R} L.$

IV.2. Rules of the grammar.

Left graphs and right graphs of the productions are shown in Figure 1. The left graph is replaced by the right graph within the host graph if the production is applicable.

We obtain 10 productions which allow the graphical operations of addition and deletion: 4 for adding each one of those different symbols, 4 for connecting a symbol already added to another one, 1 for deleting a symbol and 1 for deleting of a connection. Predicates (Π_i) express in a logical manner the restrictions in order to apply the corresponding production P_i, and have the following form. Here type(1) and type (2) refer to the types of the nodes involved in a production, as it is marked in Figure 1. To avoid unnecessary details we are going to condense the characteristics of Step 4 by "$Step_4$".

• Addition of a node

Π_1: (type(1)=C $\wedge$ (type(2)= A $\vee$ type(2)= L) $\wedge$ (adj(1) < 3) $\wedge$ $Step_4$

This means that in order for P1 to be applicable, Π_1 must be "true", i.e., the type of node 1 must be C and ($\wedge$) the type of node 2 can be A or ($\vee$) L, and ($\wedge$) the number of adjacents of node 1 must be less than 3; also it will fulfill the specifications of $Step_4$.

Π_2: (type(1)=A $\wedge$ type(2)= C $\wedge$ adj(1) < 2) $\wedge$ $Step_4$

Π_3: (type(1)=L $\wedge$ (type(2)= C $\vee$ type(2)= L $\vee$ type(2)= U))$\wedge$ $Step_4$

Π_4: (type(1)=U $\wedge$ type(2)= L) $\wedge$ $Step_4$

• Addition of a relation

Π_5: (type(1)=C adj(1)<3) $\wedge$ ((type(2)=A $\wedge$ adj(2)<2) $\vee$ type(2)=L)

Π_6: (type(1)=A $\wedge$ adj(1) < 2) $\wedge$ (type(2)=C $\wedge$ adj(2) < 3)

Π_7:(type(1)=L $\wedge$ (type(2)=C $\wedge$ adj(2)<3) $\vee$ (type(2)=L) $\vee$ (type(2)=U)

Π_8: (type(1)=U $\wedge$ type(2)=L)

• Deletion of a relation (P_{10}) and deletion of a node (P_9).

Π_9 and Π_{10}: true.

We can always delete a node or a relation.

IV.3. Control program.

Its skeleton is given by

a) Productions which need PUSH-TEST

P_1....P_7, P_9 and P_{10}; after PUSH-TEST completed, apply one of P_1....P_{10}, Z.

b) Productions which don't need PUSH-TEST

P_8 : case of P_8

- Y: P_1...P_{10}.
- N: P_1....P_7, P_9, P_{10}.

c) Matrix TB

1) types of nodes involved in productions which can modify its lower limits: C, A

2) for each symbol listed in 1) (C, A) obtain the productions in which the adjacents are modified (Fig. 4 b). Steps *a)* and *b)* let us to depict the Control Program Matrix of Fig 4 a)

• 'PT' indicates 'apply PUSH-TEST between P_i (row) and P_k (column)'

• 'YN' indicates 'apply P_k after P_i either Yes or No in P_i'

• 'Y' indicates 'apply P_k after P_i only under Yes in P_i'

• square empty: transition not possible

	1	2	3	4	5	6	7	8	9	10	Z
1	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
2	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
3	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
4	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
5	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
6	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
7	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
8	YN	YN	YN	YN	YN	YN	YN	Y	YN	YN	YN
9	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
10	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
O	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	

a)

C	P1, P5, P6, P7, P9
A	P2, P5, P6, P9

b)

Figure 4. Matrices of the control program and table TB of the example

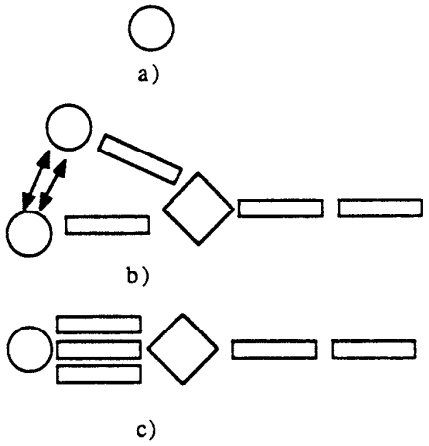

Figure 5. An example. See text for productions used.

IV.4. The grammar in use.

The CP and TB are shown in Figures 4 a) and b). In Figure 5 a) the initial graph S is a node type C and the control is in node O. Proceeding as CP indicates, we obtain Figure 5 b) from the initial graph S (C) - between parenthesis the type of node involved-, applying P_1(A), P_2(C), P_1(A), P_5, P_1(L), P_3(U), P_1(L), P_7, P_4(L) and P_3(L). Figure 5 c) is derived from Figure 5 b) applying P_{10}(L), P_{10}(C), P_{10}(A), P_{10}(A), P_1(L), P_7, P_1(L) and P_7. This is "a correct way", obviously there are other possible paths, and matrices TB and CP will assist at each step.

IV.5. A real example of this method

An effective use of this method has been done in the definition of a modeling and simulation technique [8]. The environment to be developed consisted of considering the modeling phase as the activity of graphic joining of symbols, so as to avoid the text editing phase. The structural correctness had to be ensured, and an EAPGG was designed for that purpose. In [9] the process was explained as a general framework for the development of methodologies. For the construction of the corresponding icon manipulation program the steps described in Section III have been followed, giving these results:

- 5 classes of productions (considerations about how to draw directed arcs between the icons were also taken into account)
- icon (symbol) definition: a total of 12 icons were defined
- definition of relations $\mathbb{R}$ among the icons; considering specific cases of the methodology, this led to 37 productions (and, consequently, 37 Π_i)
- definition of the adjacents and forbidden interactions: we established the adequate combinations between operands and operators

Since in the Π_i are described, in a logical form, the applicability characteristics, it was feasible to give a message about the failure of a production (a wrong connection).

The framework chosen to implement these ideas has been the object-oriented system in the Symbolics 3640 machine, according to the scheme presented in the book [4, chap. 7]. In this way the symbols and arcs are objects, and the Π_i are functions to apply on the elected nodes (through the mouse gestures or menu).

This method was integrated into the development process of a visual programming language. In that task we included the stage of designing the language, testing and implementing it. The purpose was to build a graphical software for modeling and simulation in a specific way, in such a manner that no knowledge of computer programming was required, and the system was able to provide some explanations and aids in the process of modeling process. Some restrictions come from the fact that the icons only are intended to represent mathematical expressions, and thus the semantical level does not cover the requirements of general-purpose languages.

V. CONCLUSION

The present scheme can be applied to the implementation of some specification methods for visual representations, provided they can be expressed as a set of symbols and combination rules -a recopilation of specification techniques can be found in Berztiss [3]-.

What we think is important is how the steps are structured and formalised so that other tools could be constructed in the same way, since it is configurated a specific environment for developing syntax-directed editors by APGG. Although the design of a programmed graph-grammar is fast compared to an ad-hoc programming of the graphical tool for a method, there is no aid to assist in such design. In this way we think we provide a process for considering the APGGs as a generally reusable software engineering technique, as was quoted in [12, p. 225].

In this article a quick way of developing programs for icon manipulation has been demonstrated. The application of these ideas carries on the rapid prototyping of icon manipulation programs, resulting in reliable software. Direct extensions can come from the consideration of more complex transformations and an increase in the number and type of the attributes.

REFERENCES

[1] M.B. Albizuri-Romero, "GRASE - A Graphical Syntax-Directed Editor for Structured Programming", *SIGPLAN Notices*, Vol. 19, n. 2, pp. 28-37, Feb. 1984

[2] F. Arefi et al. "Automatically Generating Visual Syntax-Directed Editors", *Comm. of the ACM*, March 1990, Vol. 33, n. 3, pp. 349-360

[3] A. Berztiss, "Formal Specification Methods and Visualization", Chap. 4 of S.K. Chang (ed.), *Principles of Visual Programming Languages*, Prentice-Hall, 1990

[4] H. Bromley and R. Lamson, "*LISP LORE: A Guide to Programming the Lisp Machine*", 2nd ed., Kluwer Academic, 1987

[5] H. Bunke, "Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation", *IEEE Trans. on Pat. Anal. and Mach. Intel.* Vol. 4, n. 6, Nov. 1982, pp. 574-582

[6] H. Bunke, "Programmed Graph Grammars", *Lect. Not. in Comp. Sci.*, Vol. 73, pp.155-166, Springer-Verlag 1979

[7] J.J. Dolado et al., "Formal Manipulation of Forrester Diagrams by Graph Grammars", *IEEE Trans. on Syst., Man and Cyb.*, Vol. 18, n. 6, pp. 981-996

[8] J.J. Dolado, "An Interface for Qualitative Simulation of System Dynamics Models" (in Spanish), *unpublished Ph. D. dissertation*, 1989, University of the Basque Country, Spain

[9] J.J. Dolado, "A Framework for the Automated Development of Graphical Mathematical Software for Systems Modeling", *Proc. of the 1989 Int. Conf. of the IEEE Syst. Man and Cyb. Soc.*, pp . 872-874

[10] M. Edel, "The Tinkertoy Graphical Programming Environment", *IEEE Trans. on Soft. Engin.*, Vol. 14, No.8, pp. 1110-1115, Aug. 1988

[11] H. Göttler, "Attributed Graph Grammars for Graphics", in *Graph-grammars and their Application to Computer Science, Lect. Not. in Comp. Sci.*, Vol.153, pp.130-142, Springer-Verlag 1983

[12] H. Göttler, "Graph Grammars and Diagram Editing", in *Graph-grammars and their Application to Computer Science, Lect. Not. in Comp. Scie.*, Vol. 291, pp. 216-231, 1987, Springer-Verlag

[13] K. Halewood and M.R. WoodWard, "NSEDIT: a Syntax-Directed Editor and Testing Tool based on Nassi-Shneiderman Charts", *Software - Practice and Experience*, 18 (10), 987-998 (1988)

[14] B.A. Myers, "Taxonomies of Visual Programming and Program Visualization", *Journal of Visual Languages and Computing*, Vol. 1, n. 1, March 1990, pp 97-123.

[15] N.C. Shu, "*Visual Programming*", Van Nostrand Reinhold, 1988