Electronic Communications of the EASST
Volume 7 (2007)

Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED 2007)

Logichart: A Prolog Program Diagram and its Layout

Yoshihiro Adachi and Yudai Furusawa

16 pages

Guest Editors: Andrew Fish, Alexander Knapp, Harald Störrle
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Logichart: A Prolog Program Diagram and its Layout

Yoshihiro Adachi[1] **and Yudai Furusawa**[2]

[1] adachi@eng.toyo.ac.jp
[2] gz0600160@toyonet.toyo.ac.jp
Department of Information and Computer Sciences
Toyo University at Kawagoe, Japan

Abstract: The layout of Logichart diagrams is first discussed. The layout condition is formalized with a layout constraint (expressions of equalities and inequalities) of tree-structured diagrams. Next, a cell placement that gives the minimum-area layout under a specific layout constraint is presented. A Logichart attribute graph grammar is then formalized. This grammar is underlain by a neighborhood controlled embedding (NCE) graph grammar whose productions are defined in order to formalize the graph-syntax rules of Logichart diagrams. Semantic rules attached to the grammar's productions are defined in such a way that they can extract the layout information needed to display a Logichart diagram by means of the attributes attached to the nodes of the graphs derived by the grammar. The semantic rules are formalized so as to obtain the Logichart diagrams of the minimum area under the above layout constraint.

Keywords: Prolog program diagrams, Attribute graph grammar, Graph layout

1 Introduction

Visualization using program diagrams can effectively facilitate the understanding, debugging, and education of programs. The Transparent Prolog Machine, a well-known Prolog visualization system [1, 2], displays the structure of a pure Prolog program as a tree with AND/OR branches (an *AND/OR tree*) and depicts the states of the various goals as symbols at its nodes. Other Prolog visualization and debugging systems, e.g., [3], also deal primarily with pure Prolog and use AND/OR trees. However, it is not easy to correlate the content of a Prolog program with that of its corresponding AND/OR tree, because the structure of the clauses of the Prolog program, and their representations in the AND/OR tree, are different.

Logichart (a Logic flowchart) is a program diagram description language that we previously developed to help visualize the execution flow of Prolog programs [4, 5]. Logichart diagrams have been developed to represent computation, which is the response of a Prolog program to a query, as an intelligible diagram. A Logichart diagram has a tree-like structure, as shown in Figure 1, with the following features: (1) the head and body goals that compose each clause are aligned horizontally, and (2) a calling goal and the heads of the clauses that it calls are aligned vertically. Feature (1) gives clauses in a Prolog program and their representations in the corresponding Logichart diagram a similar structure so that it is easier to see the correspondences between them. Feature (2) makes it easier to understand the relationships between related clauses, because they are vertically adjacent.

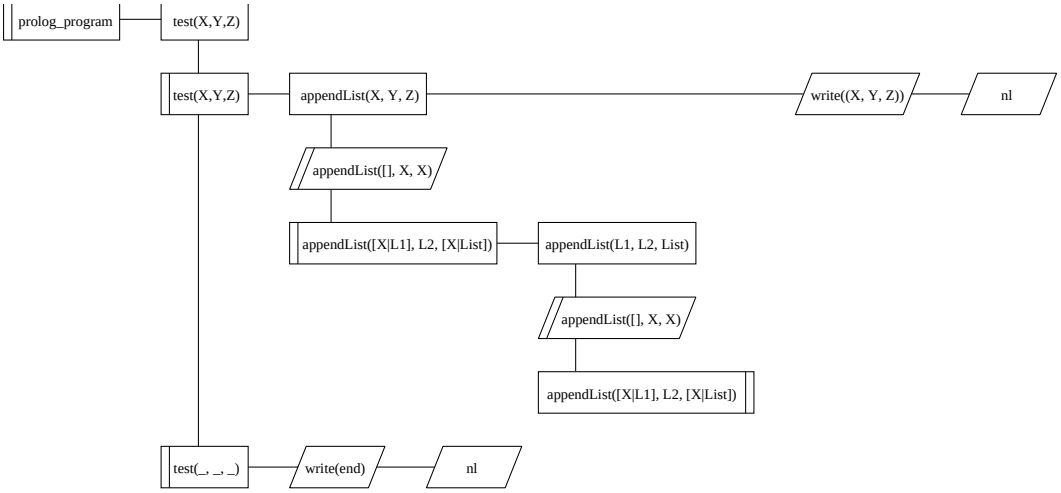

Figure 1: Example of Logichart diagrams

In this paper, we first explain the structure of Logichart diagrams and discuss their layout. The layout condition is formalized with a layout constraint (expressions of equalities and inequalities) of tree-structured diagrams. We then present a cell placement that gives the minimum-area layout of a Logichart diagram under a specific layout constraint.

A Logichart diagram has a graph structure composed of nodes (which are called cells in Logichart) and edges. The syntax rules of Logichart diagrams are therefore well formalized with a graph grammar. Furthermore, layout information (the x- and y-coordinates of each node) can be well evaluated by means of attributes attached to each node label and layout rules attached to each production of the graph grammar. Based on these viewpoints, we formalize the syntax and layout rules of Logichart diagrams based on an attribute graph grammar [6, 7], and as a result, we define the Logichart attribute graph grammar (Logichart-AGG for short). This grammar is underlain by a neighborhood controlled embedding (NCE) graph grammar that generates ordered graphs [8]. Its productions are defined in order to formalize the graph-syntax rules of Logichart diagrams, and the semantic rules attached to the grammar's productions are defined in such a way that they can extract the layout information needed to display a Logichart diagram by means of the attributes attached to the nodes of the graphs derived by the grammar. The semantic rules are formalized so as to obtain Logichart diagrams of the minimum area under the above layout constraint.

Some practical studies on formalizing the syntax of the program diagrams of imperative programming languages (e.g., Pascal, C) in terms of attribute graph grammars have already been presented [7, 9]. However, except for the authors' research, there are no known studies on formalizing Prolog program diagrams. Furthermore, our present study is the first to discuss the minimum-area layout of program diagrams under specific layout conditions and to formalize it as the semantic rules of attribute graph grammar.

2 Prolog program visualization

2.1 Logichart diagrams

Prolog is generally an interpreted language, although compiler implementations do exist. A Prolog program consists of a set of *clauses*. Each clause has a *head* and a *body*. The body consists of either no goals, one goal, or a sequence of goals. A Prolog program is executed when the interpreter is given a *query* that consists of one or more goals. The system uses its computation rule to select a goal from the query, then uses its search rule to search for a clause whose head matches the goal. If such a clause is found and called, the current query becomes (is reduced to) a new query by replacing the selected goal with the body that corresponds to the matching head. The Prolog system dynamically constructs the execution of its programs (queries) in this way.

Logichart diagrams have been developed to represent computation, which is the response of a Prolog program to a query, as an intelligible diagram [4]. A Logichart diagram is relatively easy to understand, and correspondence with the source Prolog program is clearly presented. Figure 1 shows a Logichart diagram that corresponds to the Prolog program shown below and the query '?- test(X,Y,Z).'.

```
test(X,Y,Z) :- appendList(X,Y,Z),
     write((X,Y,Z)),nl.
test(_,_,_) :- write(end),nl.
appendList([],X,X).
appendList([X|L1],L2,[X|List]) :-
     appendList(L1,L2,List).
```

2.2 Drawing constraints of Logichart diagrams

Drawing Logichart diagrams requires their cells to be laid out. This section discusses the layout conditions for Logichart diagrams.

A Logichart diagram has a rooted, binary tree structure. The layout conditions of Logichart diagrams are formalized as the layout constraints of tree-structured diagrams as follows.

Definition 1 An *L-tree-structure* T is defined by $T = (V, E, r, L_E, width, depth)$, where (V, E) is a *binary tree*, V is a set of *cells*, E is a set of *edges* and the *root* cell is $r \in V$. $L_E : E \to \{h, v\}$ is an edge labeling function. The map $width : V \to \mathbb{R}$ is the *width function* of the cells, and the map $depth : V \to \mathbb{R}$ is the *depth function* of the cells.

The horizontal length is represented by $width(p)$, which is called the *width of cell* p, and the vertical length is represented by $depth(p)$, which is called the *depth of cell* p.

Definition 2 The *placement* of an L-tree-structure, $T = (V, E, r, L_E, width, depth)$, is defined as a function, $\pi : V \to \mathbb{R} \times \mathbb{R}$. Symbols $\pi_x(p)$ and $\pi_y(p)$ denote the x-coordinate and the y-coordinate of p, respectively: $\pi(p) = (\pi_x(p), \pi_y(p))$.

Definition 3 Let $T = (V, E, r, L_E, width, depth)$ be an L-tree-structure and π be the placement of T: $D = (T, \pi)$ is called an *L-tree-structured diagram* (L-TSD for short).

L-TSDs are drawn by mapping each cell to a set of planar coordinates and joining the cells together with line segments corresponding to each edge. The x- and y-coordinates are taken along the respective horizontal and vertical directions, and the coordinates of each cell are considered to be those of its top left corner.

Definition 4 Let $T = (V, E, r, L_E, width, depth)$ be an L-tree-structure, π be the placement of T, and $D = (T, \pi)$ be an L-TSD. The *width* and the *depth of D* are defined by the following functions:

$$\begin{aligned} width(D) &\equiv max\{|\pi_x(p) + width(p) - \pi_x(q)| : \forall p, q \in V\}. \\ depth(D) &\equiv max\{|\pi_y(p) + depth(p) - \pi_y(q)| : \forall p, q \in V\}. \end{aligned}$$

Definition 5 Let $T = (V, E, r, L_E, width, depth)$ be an L-tree-structure, π be the placement of T, and $D = (T, \pi)$ be an L-TSD. The *area of D* is defined as

$$area(D) \equiv width(D) \times depth(D).$$

For cells s and t of an L-TSD, t is called a *v-child* of s if there is an edge labeled v from s to t. Similarly, t is called an *h-child* of s if there is an edge labeled h from s to t. Here, t is called a *v-descendant* of s if there is a path composed of edges labeled v. Similarly, t is called an *h-descendant* of s if there is a path composed of edges labeled h.

Here, v-subTSD with root cell s is defined as a subdiagram of an L-TSD consisting of cell s, its v-child t, an edge labeled v from s to t, and a subTSD of the L-TSD whose root cell is t. Similarly, h-subTSD with root cell s is defined as a subdiagram of an L-TSD consisting of cell s, its h-child t, an edge labeled h from s to t, and a subTSD of the L-TSD whose root cell is t. The s itself is the h-subTSD with root cell s if s has no h-child. A cell with no input edge labeled h is called a *head cell*, and a cell with an input edge labeled h is called a *goal cell*.

An edge labeled h is drawn horizontally, and an edge labeled v is drawn vertically for an L-TSD. The layout constraints for drawing L-TSDs are as follows.

Condition B_1: If t is an h-child of s, then
$\pi_y(s) = \pi_y(t)$. (Figure 2)

Condition B_2: If t is a v-child of s, then
$\pi_x(s) = \pi_x(t)$. (Figure 3)

Condition B_3: If s is a goal cell, then
$\pi_y(v\text{-child of } s) \geq \pi_y(s) + depth(s)$ + GapY. (Figure 4)

Condition B_4: If s is a head cell, then
$\pi_y(v\text{-child of } s) \geq max\{\pi_y(t) + depth(t) \mid t \text{ is a cell of } h\text{-subTSD of } s\}$ + GapY. (Figure 5)

Condition B_5: If s is a goal cell, then
$\pi_x(h\text{-child of } s) \geq max\{\pi_x(t) + width(t) \mid t \text{ is a cell of } v\text{-subTSD of } s\}$ + GapX. (Figure 6)

Condition B_6: If s is a head cell, then
$\pi_x(h\text{-child of } s) \geq \pi_x(s) + width(s)$ + GapX. (Figure 7)

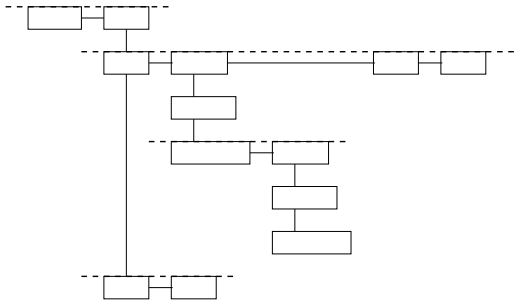

Figure 2: Layout condition B_1

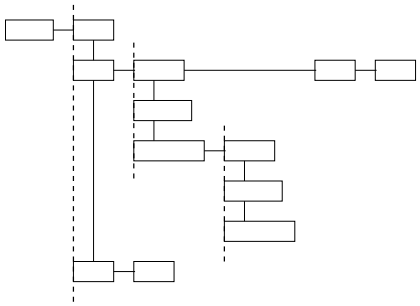

Figure 3: Layout condition B_2

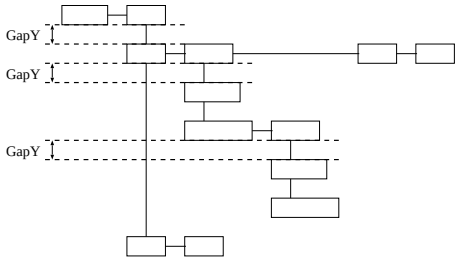

Figure 4: Layout condition B_3

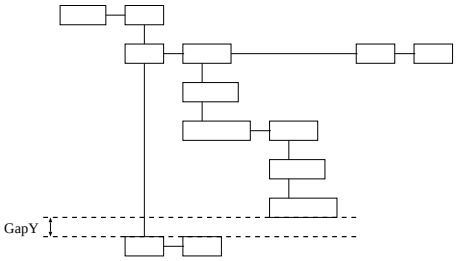

Figure 5: Layout condition B_4

The minimum-area L-TSD is obtained with the following theorem.

Theorem 1 *For any L-TSD, the placement, π, satisfying the condition, $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$, gives the minimum-area L-TSD satisfying $B_1 \wedge B_2 \wedge B_3 \wedge B_4 \wedge B_5 \wedge B_6$.*

Condition C_1: *If t is an h-child of s, then*
$\pi_y(s) = \pi_y(t)$.

Condition C_2: *If t is a v-child of s, then*
$\pi_x(s) = \pi_x(t)$.

Condition C_3: *If s is a goal cell, then*
$\pi_y(v\text{-child of } s) = \pi_y(s) + depth(s) + GapY.$

Condition C_4: *If s is a head cell, then*
$\pi_y(v\text{-child of } s) = max\{\pi_y(t) + depth(t) \mid t \text{ is a cell of h-subTSD of } s\} + GapY.$

Condition C_5: *If s is a goal cell, then*
$\pi_x(h\text{-child of } s) = max\{\pi_x(t) + width(t) \mid t \text{ is a cell of v-subTSD of } s\} + GapX.$

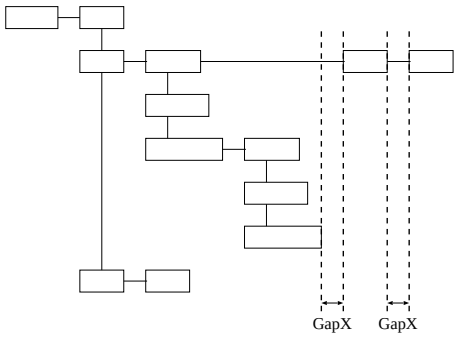

Figure 6: Layout condition B_5

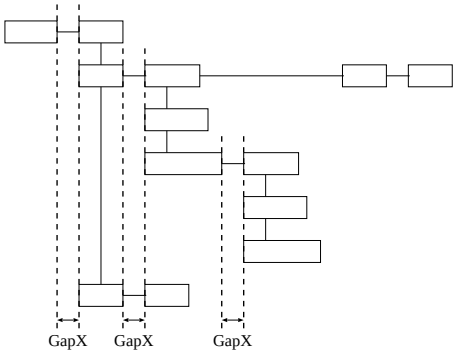

Figure 7: Layout condition B_6

Condition C_6**:** *If s is a head cell, then*
$\pi_x(h\text{-}child\ of\ s) = \pi_x(s) + width(s) + GapX.$

Proof. It is obvious that $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$ implies $B_1 \wedge B_2 \wedge B_3 \wedge B_4 \wedge B_5 \wedge B_6$. Suppose a placement, π_C, satisfies $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$. For L-TSD under the placement, π_C, cells connected with an edge labeled *h* cannot be placed closer because of the layout conditions, $B_2 \wedge B_5 \wedge B_6$. Thus, the placement, π_C, gives the layout with the minimum width. Similarly, for L-TSD under the placement, π_C, cells connected with an edge labeled *v* cannot be placed closer because of the layout conditions, $B_1 \wedge B_3 \wedge B_4$. Thus, the placement, π_C, gives the layout with the minimum depth. Consequently, the placement, π_C, gives the layout with the minimum area. □

Drawing binary trees that only have rightward-horizontal and downward-vertical segments is known as *h-v drawing*. Several studies on h-v drawing problems have been presented [10, 11]. The L-TSD drawing problem differs from these because an L-TSD's cells have a particular size (height and depth), and its layout constraints also differ from theirs.

3 Logichart-AGG

The Logichart-AGG consists of an edNCE graph grammar that generates ordered graphs and semantic rules. The edNCE grammar has formalized productions that specify the syntax rules of Logichart diagrams, and the semantic rules enable us to calculate the node coordinates of Logichart diagrams satisfying the layout constraints described in the previous section.

The Logichart-AGG specifications are very concise and consist of 13 productions associated with 88 semantic rules. The node label alphabet of the Logichart-AGG is shown in Figure 8. The edge label alphabet is $\Gamma = \Omega = \{\text{e,h,v}\}$. We introduced additional edges labeled 'h' and 'v' to the Logichart-AGG in order to regard the diagrams derived by it as L-TSDs. A Logichart diagram is obtained by removing all edges labeled 'h' and 'v' from the diagram derived by the Logichart-AGG.

The following attributes are defined to extract the node coordinates needed to make Logichart diagrams.

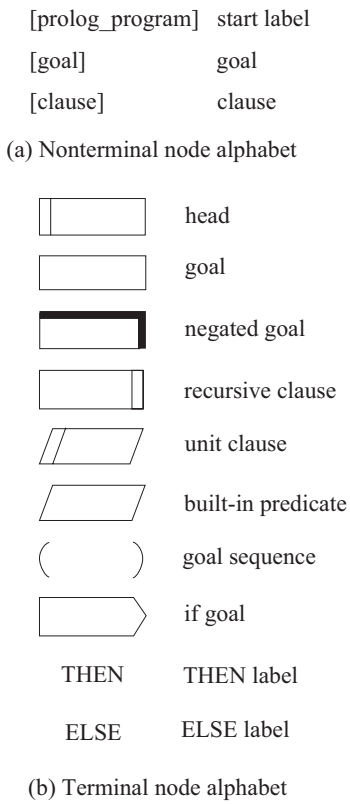

Figure 8: Node alphabets of Logichart-AGG

Inherited attributes of node labels (except the initial label):

- π_x : the x-coordinate of a node
- π_y : the y-coordinate of a node

Synthesized attributes of nonterminal node labels:

- subtree_width : the width of a subtree
- subtree_depth : the depth of a subtree

The following symbols and functions are used in the productions and semantic rules of the Logichart-AGG.

- <X> : the text string corresponding to the Prolog goal name X
- "X" : text string X
- max(x,y) : a function returns the maximum value of x and y
- get_width(x) : a function returns the width of terminal node x
- get_depth(x) : a function returns the depth of terminal node x

Some of the productions and their associated semantic rules in the Logichart-AGG are illustrated in Figures 9 to 14. The number in the lower right of each node of the productions is its identifier, and denotes its order within the nodes of the right-hand side graph. The symbol # in the connection instructions of the production definition matches any one of the node labels.

The production and semantic rules to rewrite the initial node '[prolog_ program]' are shown in Figure 9. These rules are formalized to represent queries given in Prolog syntax, and the nonterminal node 'goal' in the right-hand-side graph corresponds to the query. Semantic rules $\pi_x(1) = \text{RootX}$ and $\pi_y(1) = \text{RootY}$ mean that the x-coordinate of node 1 is 'RootX' and the y-coordinate of node 1 is 'RootY'. Semantic rule $\pi_x(2) = \text{RootX} + \text{get\_width}(1) + \text{GapX}$ means that the x-coordinate of node 2 is equal to 'RootX' plus the width of the head node labeled "prolog_program" plus the horizontal gap 'GapX'. Semantic rule $\pi_y(2) = \text{RootY}$ means that the y-coordinate of node 2 is 'RootY'. The root node "prolog_program" and the subdiagram derived from the nonterminal node 2 labeled 'goal' are aligned with a separation of 'GapX' in the horizontal direction by these semantic rules.

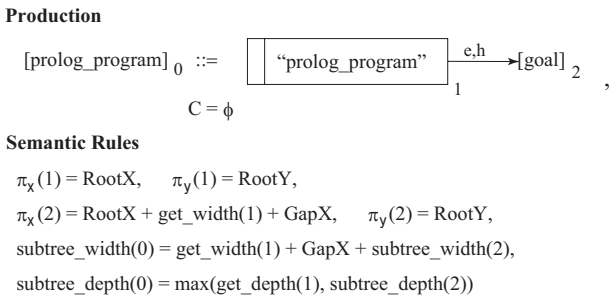

Figure 9: Rules used to rewrite initial node '[prolog_program]'

The production and semantic rules shown in Figure 10 are as formalized for the 'and' operation on Prolog goals. Semantic rule $\pi_x(2) = \pi_x(1) + \text{subtree\_width}(1) + \text{GapX}$ means that the x-coordinate of node 2 is equal to the x-coordinate of node 1 plus the width of the subdiagram derived from node 1 plus the horizontal gap 'GapX'. Therefore, goals connected by the operator 'and' are aligned with a separation of 'GapX' in the horizontal direction.

The production and semantic rules shown in Figure 11 are as formalized for the 'or' operation on Prolog goals. Semantic rule $\pi_y(2) = \pi_y(1) + \text{subtree\_depth}(1) + \text{GapY}$ means that the y-coordinate of node 2 is equal to the y-coordinate of node 1 plus the depth of the subdiagram derived from node 1 plus the horizontal gap 'GapY'. Therefore, goals connected by the operator 'or' are aligned with a separation of 'GapY' in the vertical direction.

The production and semantic rules shown in Figure 12 are formalized for the call of a goal. Semantic rule $\pi_y(2) = \pi_y(1) + \text{depth}(1) + \text{GapY}$ means that the y-coordinate of node 2 is equal to the y-coordinate of node 1 plus the depth of the cell corresponding to node 1 plus the vertical gap 'GapY'. Therefore, a calling goal and the clause heads of the goals called by it are aligned with a separation of 'GapY' in the vertical direction.

Figure 13 shows the rules for drawing negated goals. Figure 14 shows the production and semantic rules used to rewrite a nonterminal node '[goal]' with a terminal node that corresponds

Production

$[goal]_0 \quad ::= \quad [goal]_1 \xrightarrow{e,h} [goal]_2 \quad ,$

$C = \{(\#,e,e,1,in), (\#,e,e,2,out), (\#,h,h,1,in),$
$(\#,h,h,2,out), (\#,v,v,1,in), (\#,v,v,1,out)\}$

Semantic Rules

$\pi_x(1) = \pi_x(0), \quad \pi_y(1) = \pi_y(0),$
$\pi_x(2) = \pi_x(1) + subtree\_width(1) + GapX, \quad \pi_y(2) = \pi_y(1),$
$subtree\_width(0) = subtree\_width(1) + GapX + subtree\_width(2),$
$subtree\_depth(0) = \max(subtree\_depth(1), subtree\_depth(2))$

Figure 10: Rules formalized for ‘and’ operation on Prolog goals

Production

$[goal]_0 \quad ::= \quad \begin{matrix}[goal]_1 \\ \downarrow v \\ [goal]_2\end{matrix} \quad ,$

$C = \{(\#,e,e,1,in), (\#,e,e,1,out),$
$(\#,e,e,2,in), (\#,e,e,2,out),$
$(\#,h,h,1,in), (\#,h,h,1,out),$
$(\#,v,v,1,in), (\#,h,h,2,out)\}$

Semantic Rules

$\pi_x(1) = \pi_x(0), \quad \pi_y(1) = \pi_y(0),$
$\pi_x(2) = \pi_x(1), \quad \pi_y(2) = \pi_y(1) + subtree\_depth(1) + GapY,$
$subtree\_width(0) = \max(subtree\_width(1), subtree\_width(2)),$
$subtree\_depth(0) = subtree\_depth(1) + GapY + subtree\_depth(2)$

Figure 11: Rules formalized for ‘or’ operation on Prolog goals

to built-in predicates such as ‘write’, ‘nl’, and the cut ‘!’.

4 Attribute evaluation of Logichart-AGG

The definitions of an edNCE graph grammar, its leftmost derivation, an attribute graph grammar, and attribute evaluation algorithms are outlined in the appendix.

The productions and semantic rules of the Logichart-AGG are defined so as to guarantee that the simple one-pass evaluator described in Figure 16 in the appendix will evaluate all attribute instances of every c-derivation tree of the Logichart-AGG.

Theorem 2 *The Logichart-AGG is a simple one-pass.*

An L-TSD is obtained by removing all edges labeled ‘e’ from a diagram derived by the Logichart-AGG. The semantic rules of the Logichart-AGG guarantee the next theorem.

Theorem 3 *An L-TSD derived by the Logichart-AGG satisfies the layout condition* $C_1 \wedge C_2 \wedge$

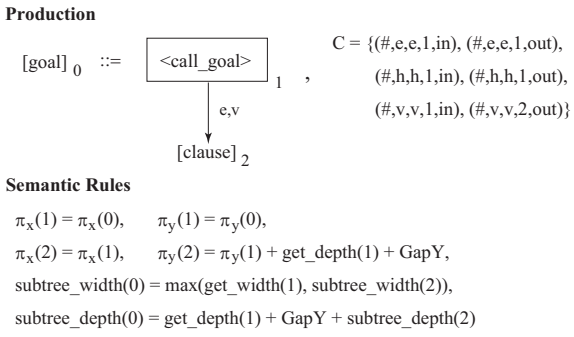

Figure 12: Rules formalized for call of goal

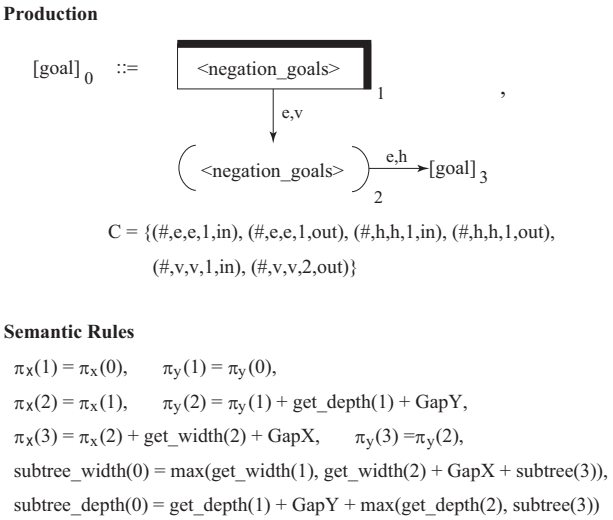

Figure 13: Rules formalized for negation of goals

$C_3 \wedge C_4 \wedge C_5 \wedge C_6$.

The above theorem states that the diagrams derived by the Logichart-AGG are the minimum-area ones under the layout constraint, $B_1 \wedge B_2 \wedge B_3 \wedge B_4 \wedge B_5 \wedge B_6$.

5 Conclusion

We presented a method of visualizing Prolog programs by means of Logichart diagrams. The layout condition for the Logichart diagrams was formalized with a layout constraint (expressions of equalities and inequalities) of tree-structured diagrams. The minimum-area layout of the Logichart diagrams was obtained under a specific layout constraint. We then formalized the syntax and layout rules of Logichart diagrams based on an attribute graph grammar, and defined

Production

$[goal]_0$ ::= <built-in_predicate>$_1$,

C = {(#,e,e,1,in), (#,e,e,1,out),
(#,h,h,1,in), (#,h,h,1,out),
(#,v,v,1,in), (#,v,v,1,out)}

Semantic Rules

$\pi_x(1) = \pi_x(0)$, $\pi_y(1) = \pi_y(0)$,
subtree_width(0) = get_width(1),
subtree_depth(0) = get_depth(1)

Figure 14: Rules formalized for built-in predicates

the Logichart-AGG. The diagrams derived by the Logichart-AGG were the minimum-area ones under the specific layout constraint.

We implemented a Prolog visualization system in complete accordance with the Logichart-AGG in the SICStus prolog [12]. The system inputs a prolog program and a query, parses them (and does not parse a graph grammar), generates an internal representation of the corresponding Logichart diagram, and displays them with Tcl/Tk. Figure 15 shows a snapshot of the Prolog visualization system.

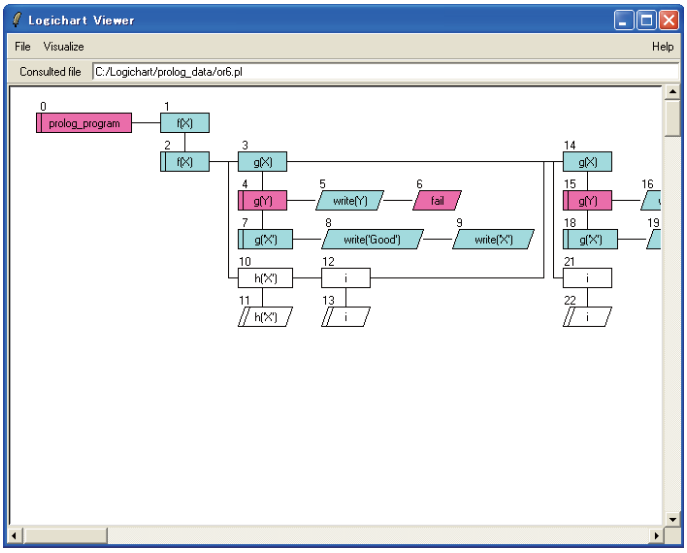

Figure 15: Snapshot of Prolog visualization system

Bibliography

[1] Eisenstadt M. and Brayshaw M.: A fine-grained account of Prolog execution for teaching and debugging, *Instructional Science*, Vol.19, No.4, pp.407-436 (1990).

[2] Brayshaw M. and Eisenstadt M.: A practical graphical tracer for Prolog, *J. Man-Machine Studies*, 35, pp.597-631 (1991).

[3] Tamir D.E.: A visual debugger for pure Prolog, *INFORMATION SCIENCES*, Vol.3, No.2, pp.127-147 (1995).

[4] Adachi Y., Imaki T., Tsuchida K. and Yaku T.: Program visualization by attribute graph grammars, *Proc. CDROM The Fundamental Conference of 15th IFIP WCC'98*, (1998).

[5] Adachi Y., Tsuchida K., Imaki T. and Yaku T.: Logichart - Intelligible Program Diagram for Prolog and its Processing System, *Electronic Notes in Theoretical Computer Science*, Volume 30, Issue 4, Elsevier Science (2000).

[6] Göttler H.: Attribute Graph Grammar for Graphics, *Lecture Notes in Computer Science*, Vol.153, pp.130-142 (1982).

[7] Nishino T.:, Attribute graph grammars with applications to Hichart editors, *Adv. Software Sci. & Tech.*, 1, pp.426-433 (1989).

[8] Engelfriet J. and Rozenberg G.: Node Replacement Graph Grammar, in *Handbook of Graph Grammars and Computing by Graph Transformation* (Rozenberg, G., eds), World Scientific, pp.1-94 (1997).

[9] Goto T., Kirishima T., Motousu N., Tsuchida K. and Yaku T.: A visual software development environment based on graph grammars, *IASTED Conf. on Software Engineering*, pp.620-625 (2004).

[10] Crescenzi P., Battista Di and Piperno A.: A note on optimal area algorithms for upward drawings of binary trees, *Computational Geometry: Theory and Applications*, 2:187-200 (1992).

[11] Eades P., Lin T. and Lin X.: Minimum size h-v drawings, *Advanced Visual Interfaces*, pp.386-394, World Scientific (1992).

[12] "Sicstus syntax", http://www.sics.se/ps/sicstus.html.

[13] Kaplan S.M. and Goering S.K.: Priority Controlled Incremental Attribute Evaluation in Attribute Graph Grammars, *TAPSOFT*, Vol.1, pp.306-336 (1989).

[14] Maneth S. and Vogler H.: Attribute Context-Free Hypergraph Grammars, *J. Automata, Languages and Combinatorics*, Vol.3, No.2, pp.105-147 (1998).

Appendix

We outline an edNCE grammar that generates ordered graphs, leftmost derivations, and c-derivation trees by referring to [8].

Attribute graph grammars were developed in the 1980s [6]. Some attribute evaluation algorithms for attribute graph grammars have been published (e.g., [13, 14]). Here, we explain an attribute graph grammar whose underlying grammar is the edNCE grammar that generates ordered graphs. We then explain an attribute evaluation algorithm on the basis of the node order of the right-hand side graphs of the edNCE grammar's productions and their c-derivation trees.

A Graph grammar and derivation

A.1 edNCE grammar

Let Σ be an alphabet of nodes and Γ be an alphabet of edges. A *graph* over Σ and Γ is a 3-tuple $H = (V, E, \lambda)$, where V is the finite nonempty set of nodes, $E \subseteq \{(v, \gamma, w) \mid v, w \in V, v \neq w, \gamma \in \Gamma\}$ is the set of edges, and $\lambda : V \to \Sigma$ is the node labeling function. The components of a graph, H, are denoted by V_H, E_H, and λ_H. Two graphs H and K are *isomorphic* if there is a bijection $f : V_H \to V_K$ such that $E_K = \{(f(v), \gamma, f(w)) \mid (v, \gamma, w) \in E_H\}$ and, for all $v \in V_H$, $\lambda_K(f(v)) = \lambda_H(v)$. The set of all graphs over Σ and Γ is denoted as $GR_{\Sigma,\Gamma}$.

Definition 6 An *edNCE grammar* is a tuple $G = (\Sigma, \Delta, \Gamma, \Omega, P, S)$, where

(1) Σ is the alphabet of node labels.

(2) Δ is the alphabet of terminal node labels.

(3) Γ is the alphabet of edge labels.

(4) Ω is the alphabet of final edge labels.

(5) P is the finite set of productions. A production has the form '$X \to (D, C)$', where

  (5.1) $X \in \Sigma - \Delta$ and $D \in GR_{\Sigma,\Gamma}$.

  (5.2) $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_D \times \{\text{in, out}\}$ is the connection relation of p and each element $(\sigma, \beta, \gamma, x, d)$ $(\sigma \in \Sigma, \beta, \gamma \in \Gamma, x \in V_D, d \in \{\text{in, out}\})$ of C is a connection instruction of p.

(6) $S \in \Sigma - \Delta$ is the initial nonterminal node label. The *initial graph*, H_s, is a graph that consists of a single node labeled S and has no edge.

Two productions $p_1 : X_1 \to (D_1, C_1)$ and $p_2 : X_2 \to (D_2, C_2)$ are called isomorphic if $X_1 = X_2$ and there is an isomorphism f from D_1 to D_2 and $C_2 = \{(\sigma, \beta, \gamma, f(x), d) \mid (\sigma, \beta, \gamma, x, d) \in C_1\}$. By copy$(p)$, we denote the set of all productions that are isomorphic to a production, p, in P. An element of copy(p) is called a *production copy* of p. $copy(P) = \bigcup_{p \in P} \text{copy}(p)$.

The process of graph rewriting in an edNCE graph grammar is defined as follows. Let a given graph (host graph) be H and let v be a nonterminal node of H. Let node v be labeled X, and let $p' : X \rightarrow (D', C')$ be a production copy of some $p \in P$ such that D' and H are mutually disjoint.

(1) Remove node v and all edges that are incident with v from H. Let the resulting graph (rest graph) be H^-.

(2) Put D' (daughter graph) in H^-.

(3) Establish new edges between certain nodes of D' and certain nodes of H^- in a way specified by the connection instructions in C'. This is called the embedding of D' in H^-. Let the resulting graph be H'.

The meaning of an instruction, $(\sigma, \beta, \gamma, x, \text{in}) \in C'$, is as follows: if there is an edge with label β from a node, $w \in V_H - \{v\}$, with label σ to node v, then the embedding process should establish an edge with label γ from node w to node $x \in V_{H'}$. Also, this is similarly the case for $(\sigma, \beta, \gamma, x, \text{out})$ instead of $(\sigma, \beta, \gamma, x, \text{in})$, where 'in' refers to the incoming edges of v and 'out' to the outgoing edges of v.

Let H and H' be graphs in $GR_{\Sigma,\Gamma}$. If H is transformed into H' by the application of a production copy, p', as described above, then we write $H \Rightarrow_{v,p'} H'$ and call it a *derivation step*. A sequence of derivation steps $H_0 \Rightarrow_{v_1,p'_1} H_1 \Rightarrow_{v_2,p'_2} \cdots \Rightarrow_{v_n,p'_n} H_n$ is called a *derivation*. A derivation, $H_0 \Rightarrow_{v_1,p'_1} H_1 \Rightarrow_{v_2,p'_2} \cdots \Rightarrow_{v_n,p'_n} H_n$, $n \geq 0$, is *creative* if the graphs, H_0 and D'_i, $1 \leq i \leq n$, are mutually disjoint, where $\text{rhs}(p'_i) = (D'_i, C'_i)$. We will restrict ourselves to creative derivations. We write $H \Rightarrow^* H'$ if there is a creative derivation as above, with $H_0 = H$ and $H_n = H'$. A *sentential form* of G is a graph, H, such that $H_S \Rightarrow^* H$ for some initial graph H_S. A set of sentential forms is denoted by $\mathscr{S}(G)$. The *graph language generated by* G is $\mathscr{L}(G) = \{[H] \mid H \in GR_{\Delta,\Omega}$ and $H_S \Rightarrow^* H$ for some initial graph $H_S\}$. For a graph, H, the set of all graphs isomorphic to H is denoted as $[H]$.

A.2 Leftmost derivation

The leftmost derivation is defined as follows. Let a given ordered graph (host graph) be H and let v be the first nonterminal node within the node order of H. Let node v be labeled X, and let $p' : X \rightarrow (D', C')$ be a production copy of some $p \in P$ such that D' and H are mutually disjoint.

(1) Remove node v and all edges that are incident with v from H. Let the resulting graph (rest graph) be H^-.

(2) Put D' (daughter graph) in H^-.

(3) Establish new edges between certain nodes of D' and certain nodes of H^- in the way specified by the connection instructions in C'. Let the resulting graph be H'.

(4) If the nodes of H are ordered as $(v_1, \cdots, v_h)$ with $v = v_i$, and those of D' are ordered as $(w_1, \cdots, w_d)$, then the order, H', is $v_1, \cdots, v_{(i-1)}, w_1, \cdots, w_d, v_{(i+1)}, \cdots, v_h$.

Let the initial graph, H_s, be an ordered graph that consists of a single node labeled S. If an ordered graph, H, is transformed into an ordered graph, H', by the application of a production

copy, p', as described above, then we write $H \Rightarrow_{p'} H'$ and call it a *leftmost derivation step*. A sequence of leftmost derivation steps $H_0 \Rightarrow_{v_1,p'_1} H_1 \Rightarrow_{v_2,p'_2} \cdots \Rightarrow_{v_n,p'_n} H_n$ is called a *leftmost derivation*.

We write $H \Rightarrow^*_{lm} H'$ if there is a leftmost derivation from H to H'. The *graph language leftmost generated by* G is $\mathscr{L}_{lm}(G) = \{[H] \mid H \in GR_{\Delta,\Omega}$ and $H_S \Rightarrow^*_{lm} H$ for some initial graph $H_S\}$, where the abstract graph, $[H]$, no longer involves an order.

A.3 Derivation tree

Derivation trees are rooted, ordered trees. Each vertex of a tree has directed edges to each of its k children, $k \geq 0$, and the order of the children is indicated by labeling the edges as $1, \cdots, k$.

Definition 7 ([8]) Let $G = (\Sigma, \Delta, \Gamma, \Omega, P, S)$ be an edNCE grammar. A *c-labeled derivation tree* of G is a rooted, ordered tree t whose vertices are labeled by production copies in $copy(P)$, such that

(1) the right-hand sides of all the production copies that label the vertices of t are mutually disjoint, and do not contain the root of t as a node, and

(2) if vertex v of t has label $X \to (D,C)$, then the children of v are the nonterminal nodes of D, and their order in t is the same as their order in D; moreover, for each child w, the left-hand side of the label of w in t equals its label in D.

If the root of a derivation tree, t, is labeled with production $X \to (D,C)$, then we also say that t is a derivation tree *for* X (note that not necessarily $X = S$).

B AGG and attribute evaluation

B.1 Definition of AGG

An AGG in this work is defined by referring to [7].

Definition 8 An *attribute graph grammar* (*AGG*) is a tuple $AG = < G, A, F >$, where

(1) $G = (\Sigma, \Delta, \Gamma, \Omega, P, S)$ is an edNCE grammar that generates order graphs and we call it an *underlying graph grammar* of AG.

(2) Each node label $X \in \Sigma$ of G has two disjoint finite sets $Inh(X)$ and $Syn(X)$ of *inherited* and *synthesized attributes*, respectively. We denote the set of all attributes of nonterminal node symbol X by $A(X) = Inh(X) \cup Syn(X)$. $A = \bigcup_{X \in \Sigma} A(X)$ is called the *set of attributes* of G. We assume that $Inh(S) = \phi$, and that for each $X \in \Delta$ $Syn(X) = \phi$. An attribute of X is denoted by $a(X)$, and the set of possible values of a is denoted by $\mathscr{V}(a)$.

(3) Associated with each production $p : X_0 \to (D,C)$ is a set F_p of *semantic rules*, which define all the attributes in $Syn(X_0) \cup Inh(X_1) \cup \cdots \cup Inh(X_n)$ where $V_D = \{v_1, \cdots, v_n\}$ and $\lambda_D(v_i) = X_i, 1 \leq i \leq n$. A semantic rule defining an attribute, $a(X_k), 0 \leq k \leq n$, has form

$a(X_k) := f(a_1(X_{i_1}),\ldots,a_m(X_{i_m}))$, where f is a mapping, $\mathscr{V}(a_1(X_{i_1})) \times \ldots \times \mathscr{V}(a_m(X_{i_m}))$ into $\mathscr{V}(a(X_k))$. In this situation, we say that $a(X_k)$ depends on $a_1(X_{i_1}),\cdots,a_m(X_{i_m})$ in p. The set, $F = \bigcup_{n\in P} F_p$, is called the *set of semantic rules* of AG.

B.2 Attribute evaluation algorithm

The graph nodes generated by AGGs have no natural linear ordering. Therefore, to evaluate attributes properly, we need to first decide the order in which the vertices of the c-derivation trees of the AGG are traversed. To do this, we use an edNCE grammar that generates ordered graphs as the underlying grammar of the AGG. The vertices of c-derivation trees are then traversed in the node order of the right-hand side graphs of the edNCE grammar's productions.

An AGG G is a *simple one-pass* if the simple-pass evaluator given in Figure 16 evaluates all attribute instances of every c-labeled derivation tree for the S of G. The simple-pass evaluator traverses the vertices of a c-derivation tree t in depth-first and then from left to right.

```
procedure EVALUATE(v0); vertex v0;
{Let the label of vertex v0 be X → (D,C) and let the nodes
of D be ordered as v1, v2, ··· , vn.}
    begin
        compute all inherited attributes of X;
        for i := 1 to n do
            begin
                if vi is a nonterminal node then EVALUATE(vi);
                else compute all attributes of λD(vi);
            end
        compute all synthesized attributes of X;
    end

Simple one-pass
    begin
        EVALUATE(root);
    end
```

Figure 16: Simple one-pass evaluator