Visual Software Development Environment Based on Graph Grammars

Takaaki GOTO†(a), Student Member, Kenji RUISE††, Takeo YAKU†††, and Kensei TSUCHIDA††††, Members

SUMMARY In software design and development, program diagrams are often used for good visualization. Many kinds of program diagrams have been proposed and used. To process such diagrams automatically and efficiently, the program diagram structure needs to be formalized. We aim to construct a diagram processing system with an efficient parser for our program diagram Hichart. In this paper, we give a precedence graph grammar for Hichart that can parse in linear time. We also describe a parsing method and processing system incorporating the Hichart graphical editor that is based on the precedence graph grammar.

key words: program diagrams, attribute graph grammar, graph editor, Hichart, precedence graph grammar, SVG

1. Introduction

In software design and development, program diagrams are often used for good visualization. Many kinds of program diagrams, such as Hichart (Hierarchical flowchart language), PAD (Problem Analysis Diagram), HCP (Hierarchical and Complex Description Chart) and SPD (Structured Programming Diagram), have been used in software development [1], [2]. Software development using these program diagrams is on the increase, hence a means of data exchange between different CASE-tools for different program flow charts is desirable. In fact, DXL (Diagram eXchange Language for tree-structured charts) was specified in the 1997 ISO for this purpose [3].

Large-scale program diagrams need to be parsed efficiently and drawn automatically, and to enable this, the program diagram structure needs to be formalized. A graph grammar is a formal method that strictly defines the mechanisms such as generation and parsing. Research on graph grammars includes studies by Frank [4], Nagl [5], Rozenberg [6], and others. Various programming environments based on graph grammars have been proposed [6]. For example, DiaGen targets general graphs, while IPSEN targets program semantics.

Our research adopts the program diagram Hichart. Hichart is a program diagram methodology that was introduced by Yaku and Futatsugi [7]. It has three key features: (1) A diagram is a tree-flowchart that has the flow control lines of a Neumann program flowchart, (2) The nodes of the different functions in a diagram are represented by differently shaped cells, and (3) The hierarchy of the data structure represented by a diagram and the control flow are simultaneously displayed on a plane that distinguishes it from other program diagram methodologies. There has been a substantial amount of research devoted to Hichart. A prototype formulation of attribute graph grammar for Hichart was reported in [8]. This grammar consists of Hichart syntax rules, which use a context free graph grammar [9], and semantic rules for layout. The authors have been developing a software development environment based on graph theory, which includes graph drawing theory and graph grammars [2], [10]. So far, we have developed bidirectional translators that translate Pascal, C, or DXL source into Hichart and which translate Hichart into Pascal, C, or DXL [2], [10]. For instance, HiChart Graph Grammar (HCGG) is introduced in [11]. HCGG is an attribute graph grammar with an underlying graph grammar based on edNCE graph grammar [6], and it is intended for use with DXL. It has the problem that the HCGG is not considered to parse efficiently. HCPGG (Hichart Precedence Graph Grammar) is introduced in [12]. It has precedence relations, and it is used for efficient parsing. It also has a problem regarding precedence conflicts.

The task of processing large-scale program diagrams needs an efficient parser, and as yet, no research that we know of has come up with an adequate way to make diagrams of large programs automatically. A framework that can ensure precise behavior and proof of parsing efficiency is also needed.

Distributed software development is on the increase, so it has become necessary for software specification documents to be shared on the Web. With regard to Web documents, XML and SVG have been proposed as standard document and graphical formats for the Web. Scalable Vector Graphics (SVG) [13] is a W3C Recommendation and a language for describing two-dimensional graphics and graphical applications in XML. SVG can display graphical objects on any readily available Web browser. With these formats, users can share documentation including graphical objects on the Web. We reported on automatic generation of SVG...
files and incorporated the generation method into a graphical editor for Hichart by using attribute graph grammars [14].

In this paper, we construct an efficient parser for Hichart by attribute graph grammar. We checked the current Hichart/DXL graph grammar and found it is not a precedence graph grammar. Consequently, we revised the graph grammar to precedence graph grammar. Accordingly, we use a precedence relation for Hichart graph grammar to develop an efficient parser for diagrams. We propose an algorithm for the precedence parser and its processing system. The processing system is a graphical editor supporting structure–free or free-hand editing with a parser. We also describe automatic generation of an SVG file that can be used to draw aesthetic diagrams based on an attribute evaluation of the derivation tree generated by the parser.

This paper is organized as follows: Section 2 give preliminaries. In Sect. 3, we review the Hichart/DXL. Section 4 describes the graph grammar for Hichart/DXL. In Sect. 5, we describe the parsing of the graph grammar for Hichart/DXL. Section 6 explains the Hichart Editor. In Sect. 7, we give discussion. We conclude in Sect. 8.

2. Preliminaries

Here, we review the notations and definitions.

Definition 1: ([6], [15]) Let \( \Sigma \) be an alphabet of node labels and \( \Gamma \) be an alphabet of edge labels. A graph over \( \Sigma \) and \( \Gamma \) is a tuple \( H = (V, E, \lambda) \), where \( V \) is the finite set of nodes, \( E \subseteq \{v, y, w | v, w \in V, v \neq w, y \in \Gamma\} \) is the set of edges, and \( \lambda : V \rightarrow \Sigma \) is the node labeling function. \( E(v, w) \) is the label tuple of two nodes \( v, w \in V \) is \( \text{lab}(v, w) = (\lambda(v), E(v, w), E(w, v), \lambda(w)) \).

In this paper, we consider directed graphs without loops. A node on a graph has attributes such as coordinates and cell size.

Definition 2: ([6]) Two graphs \( H \) and \( K \) are isomorphic if there is a bijection \( f : V_H \rightarrow V_K \) such that \( E_K = \{f(v), y, f(w) | (v, y, w) \in E_H\} \) and for all \( v \in V_H \), \( \lambda_K(f(v)) = \lambda_H(v) \).

Definition 3: ([6]) The set of all concrete graphs over \( \Sigma \) and \( \Gamma \) is denoted as \( GR_{\Sigma, \Gamma} \). The set of all abstract graphs is denoted as \( [GR_{\Sigma, \Gamma}] \). A subset of \( [GR_{\Sigma, \Gamma}] \) is called a graph language.

Definition 4: ([6]) A graph with (neighbourhood controlled) embedding over \( \Sigma \) and \( \Gamma \) is a pair \( (H, C) \) with \( H \in GR_{\Sigma, \Gamma} \) and \( C \subseteq \Sigma \times \Gamma \times \Sigma \times (\text{in}, \text{out}) \). \( C \) is the connection relation of \( (H, C) \), and each element \((\sigma, \beta, y, x, d)\) of \( C \) (with \( \sigma \in \Sigma, \beta \in \Gamma, x \in V_H, \) and \( d \in \{\text{in}, \text{out}\} \) is a connection instruction of \( (H, C) \). A connection instruction \((\sigma, \beta, y, x, d)\) will always be written as \((\sigma, \beta/y, x, d)\). Two graphs with embedding \((H, C_H)\) and \((K, C_K)\) are isomorphic if there is an isomorphism \( f \) from \( H \) to \( K \) such that \( C_K = \{(\sigma, \beta/y, f(x), d) | (\sigma, \beta/y, x, d) \in C_H\} \). The set of all graphs with embedding over \( \Sigma \) and \( \Gamma \) is denoted as \( GRE_{\Sigma, \Gamma} \).

Definition 5: ([6]) An edNCE graph grammar is a tuple \( GG = (\Sigma, \Delta, \Gamma, \Omega, P, S) \), where \( \Sigma \) is the alphabet of node labels, \( \Delta \subseteq \Sigma \) is the alphabet of terminal node labels, \( \Gamma \) is the alphabet of edge labels, \( \Omega \subseteq \Gamma \) is the alphabet of final edge labels, \( P \) is the finite set of productions, and \( S \in \Sigma - \Delta \) is the initial nonterminal. A production is of the form \( X \rightarrow (D, C) \) where \( X \) is a nonterminal node label, \( D \) is a graph over \( \Sigma \) and \( \Gamma \), and \( C \subseteq \Sigma \times \Gamma \times \Sigma \times \Gamma \times \{\text{in}, \text{out}\} \) is the connection relation which is a set of connection instructions. A pair \((D, C)\) is a graph with embedding over \( \Sigma \) and \( \Gamma \).

Definition 6: ([6]) A copy\((P)\) denotes the infinite set of all productions that are isomorphic to a production in \( P \); an element of copy\((P)\) is called a production copy of \( GG \).

Definition 7: ([6]) Let \((H, C_H)\) and \((D, C_D)\) be two graphs with embedding, in \( GRE_{\Sigma, \Gamma} \), such that \( H \) and \( D \) are disjoint, and let \( v \) be a node of \( H \). The substitution of \((D, C_D)\) for \( v \) in \((H, C_H)\), denoted as \((H, C_H)[v/\{D, C_D]\}], is the graph with embedding \((V, E, A, C)\) in \( GRE_{\Sigma, \Gamma} \) such that \( V = (V_H - \{v\}) \cup V_D \), \( E = \{(x, y, v) \in E_H | x \neq v, y \neq v \} \cup E_D \), \( \phi((w, y, x) | \exists \beta \in \Gamma : (v, \beta, y, x, \text{in}) \in E_D) \cup((x, y, w) | \exists \beta \in \Gamma : (v, \beta, y, w) \in E_D) \cup((x, \beta/y, y, x, \text{out}) \in C_D) \), \( \lambda(x) = \lambda_H(x) \) if \( x \in V_H - \{v\} \), and \( \lambda(x) = \lambda_D(x) \) if \( x \in V_D \), \( C = \{(\sigma, \beta/y, x, d) \in C_H | x \neq v \} \cup((\sigma, \beta, y, x, d) | \exists y \in \Gamma : (\sigma, \beta/y, v, d) \in C_H) \).

Definition 8: ([6], [15]) Let \( G = (\Sigma, \Delta, \Gamma, \Omega, P, S) \) be an edNCE graph grammar. Let \( H_{i-1} = (V_{H_{i-1}}, E_{H_{i-1}}, \lambda_{H_{i-1}}) \) and \( H_i = (V_{H_i}, E_{H_i}, \lambda_{H_i}) \) be graphs in \( GRE_{\Sigma, \Gamma} \). In addition, let \( v_i \in V_{H_{i-1}} \), and \( p_i : X \rightarrow (D'_i, C'_i) \in P \) be a production copy of \( G \) such that \( D'_i \) and \( H_{i-1} \) are disjoint. \( s_i = (p_i/v_i, D'_i, b'_i) \) is a derivation specification of \( G \) if \( p_i \in \text{copy}(P) \), \( \lambda_{H_{i-1}}(v_i) = X \), \( D'_i \equiv D, b'_i : V_{D'_i} \rightarrow V_D \).

We write \( H_{i-1} \rightarrow_{v_i, p_i} H_i \), or just \( H_{i-1} \rightarrow H_i \), if \( \lambda_{H_{i-1}}(v_i) = X \) and \( H_i = H_{i-1}[v_i/(D'_i, C'_i)] \). \( H_{i-1} \rightarrow H_i \) is called a derivation step, and a sequence of such derivation steps is called a derivation.

Figure 1 shows an example of an application of a production. In the Fig. 1 \( H = (V_H, E_H, \lambda_H) \) is a graph with \( V_H = [v_1, v_2], E_H = [(v_1, v_2), (v_2, v_3)], \lambda_H(v_1) = a \) and \( \lambda_H(v_2) = a \). The production copy \( p' \) is as follows: \( p' : X \rightarrow (D', C') \) where \( X = \lambda_H(v_2), D' \equiv (V_{D'}, E_{D'}, \lambda_{D'}) \) such that \( V_{D'} = [v_3, v_4], E_{D'} = [(v_3, v_4)], \lambda_{D'}(v_3) = b \), \( \lambda_{D'}(v_4) = Y \) and \( C' = [(a, a/\beta, v_3, \text{in})] \).

The production copy \( p' \) is applied to the node \( v_2 \) of \( H \). After that we get the graph \( H' = (V_{H'}, E_{H'}, \lambda_{H'}) \) where \( V_{H'} = [v_1, v_3, v_4], E_{H'} = [(v_1, v_3), (v_3, v_4)], \lambda_{H'}(v_1) = a, \lambda_{H'}(v_3) = b, \lambda_{H'}(v_4) = Y \).
The following definitions pertain to the attribute graph grammar.

**Definition 9:** ([16]) An Attribute edNCE Graph Grammar is a tuple $AGG = (GG, Att, F)$, where

1. $GG = (\Sigma, \Delta, \Gamma, \Omega, P, S)$ is called an underlying graph grammar of AGG. Each production $p \in P$ is denoted by $X \rightarrow (D, C)$. $Lab(D)$ denotes the set of all occurrences of the node labels in the graph $D$.

2. Each node symbol $Y \in \Sigma$ of $GG$ has two disjoint finite sets $Inh(Y)$ and $Syn(Y)$ of inherited and synthesized attributes, respectively. The set of all attributes of symbol $X$ is defined as $Att(X) = Inh(X) \cup Syn(X)$. $Att = \bigcup_{X \in G} Att(X)$ is called the set of attributes of AGG. We assume that $Inh(S) = \emptyset$. An attribute $a$ of $X$ is denoted by $a(X)$, and the set of possible values of $a$ is denoted by $V(a)$.

3. Associated with each production $p = X_0 \rightarrow (D, C) \in P$ is a set $F_p$ of semantic rules which define all the attributes in $Syn(X_0) \cup \bigcup_{X \in Lab(D)} Inh(X)$. A semantic rule defining an attribute $a_0(X_0)$ has the form $a_0(X_0) := f(a_1(X_1), \ldots, a_m(X_m))$. Here $f$ is a mapping from $V(a_1(X_1)) \times \cdots \times V(a_m(X_m))$ into $V(a_0(X_0))$. In this situation, we say that $a_0(X_0)$ depends on $a_j(X_j)$ for $j, 0 \leq j \leq m$ in $p$. The set $F = \bigcup_{p \in P} F_p$ is called the set of semantic rules of $G$.

Nodes generated by our graph grammar have attributes such as coordinates and cell sizes. Attribute values are calculated by evaluating attributes according to semantic rules on the obtained derivation tree.

Figure 2 is an example of a production with semantic rules. There are two attribute types, inherited attribute and synthesized attribute. Inherited attributes are calculated from the root node, and the root node’s values are inherited to the leaf nodes. The synthesized attributes are calculated from leaf nodes to the root node. For example, in Fig. 2, $x$ is an inherited attribute and the attribute $y$ is a synthesized attribute.

The following describes the precedence graph grammar. We apply the framework of precedence graph grammars devised by Kaul to the edNCE graph grammars of Rozenberg.

Let $D = (G_{i-1} \rightarrow G_i \mid 1 \leq i \leq n)$, $n \in \mathbb{N}$, $s_i = (p_i, L_i, R_i, b_i)$ be a derivation sequence. $s_i$ precedes $s_j$ if $L_j$ is an induced subgraph of $R_i$, $1 \leq i, j \leq n$. The reflexive and transitive closure of this relation is denoted as $\leq_D$. Note that relations $\prec_D$, $\succ_D$ and $=D$ can be defined by using $\leq_D$.

Let $s_D(v) = s_i$ if $v \in V_{R_i}$, $1 \leq i \leq n$.

**Definition 10:** ([15]) The derivation order of the nodes in $G_a$ is as follows: $v \prec w \iff s_D(v) < s_D(w)$, $v \prec_D w \iff s_D(v) \succ_D s_D(w)$, $v \succ w \iff s_D(v) > s_D(w)$, $v \succ_D w \iff s_D(v) = s_D(w)$, where $v, w \in V_{G_a}$ and node $v$ and $w$ is adjacent.

**Definition 11:** ([15]) The derivation specifications $s_i, s_j$ are incomparable if neither $s_i \leq_D s_j$ nor $s_j \leq_D s_i$.

**Definition 12:** ([15]) $v \succ_D w$ if $s_D(v), s_D(w)$ are incomparable. $\preceq, \leq, \succ, \geq$ are the precedence relations between nodes. The precedence relations between labels, $R_0$, $\Theta \in \{\preceq, \leq, \succ, \geq\}$ is the set of all $lab_G(v, w)$ s.t. there is a derivable graph $G$, $v, w \in V_G$, $v \Theta w$.

Figure 3 shows an example calculation of precedence relations. In Fig. 3 $G_a = (V_{G_a}, E_{G_a}, \lambda_{G_a})$ is a graph where $V_{G_a} = \{n_2, n_3, n_4\}$, $E_{G_a} = \{(n_2, n_3), (n_2, n_4)\}$, $\lambda_{G_a}(n_2) = a$, $\lambda_{G_a}(n_3) = B$, and $\lambda_{G_a}(n_4) = C$. We construct a production copy $p'_4$ of $p_4$ as follows: $p'_4 : X \rightarrow (D'_p, C'_p)$ where $X'_p = \lambda_{G_a}(n_4)$, $D'_p = (V_{p'}, E'_{p'})$ such that $V_{p'} = \{n_5, n_6\}$, $E'_{p'} = \{(n_5, n_6)\}$, $X_{p'}(n_5) = b$, $X_{p'}(n_6) = X$ and $C_{p'} = \{(a, a/b, n_5, n_6)\}$. Then we apply the production copy $p'_4$ to node $n_4$ of $G_a$. Finally the graph $G_b$ is obtained. The derivation specification of this application is $s_4 = (p_4, L_\alpha, R_\alpha, b_\alpha)$ where $L_\alpha = \lambda_{G_a}(n_4)$, $R_\alpha$ is isomorphic
to the right hand side of production $p_\alpha$ and $\tilde{b}_\alpha(n_3) = v_1$, $\tilde{b}_\alpha(n_5) = v_2$.

Now let us consider the precedence relations between nodes and labels. These relations are computed according to Definition 12. First, we obtain a derivation specifications $s_x$ that generate nodes $n_5$ and $n_6$. $S_D(n_5) = s_\alpha$ and $S_D(n_6) = s_\beta$. Nodes $n_5$ and $n_6$ are generated by the same derivation specification $s_x$. Therefore, the precedence relation between node $n_5$ and $n_6$ is $n_5 \prec n_6$. Moreover, the precedence relation between the node labels of $n_5$ and $n_6$ is lab($n_5$, $n_6$) = ($b$, $\gamma$, $0$, $X$) $\in R_\prec$.

**Definition 13:** ([15]) A graph grammar $GG$ is confluent if for all derivable graphs $G_1$, $G_2$, $G_3$ and incomparable derivation specifications $s_1$, $s_2$ with $D = (G_1 \rightarrow G_2)$, $D = (G_1 \rightarrow G_3)$ there is a derivable graph $G_4$ such that $D = (G_2 \rightarrow G_4)$ and $D = (G_3 \rightarrow G_4)$ are derivation steps. Let $p : X \rightarrow (D, C)$ be a production, $I_\Sigma =_{\text{def}} \{(\sigma, \beta, \gamma, x, \text{in}) \mid \sigma \in \Sigma, \beta, \gamma \in \Gamma, x \in V_D\}$. $O_\Sigma =_{\text{def}} \{(\sigma, \beta, \gamma, x, \text{out}) \mid \sigma \in \Sigma, \beta, \gamma \in \Gamma, x \in V_D\}$. $p$ is symmetric if $I_\Sigma = I_{\Sigma(x)}$, $O_\Sigma = O_{\Sigma(x)}$ for automorphisms $a : V_D \rightarrow V_D, x \in V_D$. $GG$ is symmetric if all productions in $GG$ are symmetric. $GG$ is uniquely invertible if every derivation step can be inverted uniquely only by inspection of the substituted subgraph and its direct neighbourhood. A nonterminal $B$ is called reflexive if $B$ can be derived from $B$ in at least one step. A precedence conflict is a label tuple $t$ that occurs in more than one precedence relation. □

**Definition 14:** ([15]) A graph grammar that is confluent, symmetric, and uniquely invertible, and has no reflexive nonterminals and no precedence conflicts, is called a precedence graph grammar. □

3. Hichart/DXL

Hichart [7] is a program diagram having the following characteristics: (1) A diagram is a tree-flowchart that has the flow control lines of a Neumann program flowchart; (2) The nodes of the different functions in a diagram are represented by differently shaped cells; (3) The hierarchy of the data structure represented by a diagram and the control flow are simultaneously displayed on a plane.

The Diagram eXchange Language for tree-structured charts DXL is specified in the 1997 ISO [3]. The primary purpose of DXL is to provide a means of exchanging data between different CASE-tools for different program flow charts. This purpose allows various CASE-tools to use specifications made in the past.

Figure 4 shows an example of an Hichart for DXL that describes an example of DXL code on ISO/IEC 14568 [3].

4. Graph Grammar for Hichart/DXL

4.1 Attribute Graph Grammar for Hichart/DXL

In this section, we describe an attribute graph grammar that defines Hichart for DXL (Hichart/DXL) diagrams. It is called PGGHD (Precedence Graph Grammar for Hichart/DXL), and it is defined using attribute edNCE graph grammar [17].

**Definition 15:** A graph grammar for Hichart/DXL is a tuple $\text{PGGHD} = (\Sigma_{HD}, \Delta_{HD}, \Omega_{HD}, P_{HD}, S_{HD})$, where $\Sigma_{HD}$ is the alphabet of node labels, $\Delta_{HD} \subseteq \Sigma_{HD}$ is the alphabet of terminal node labels, $\Omega_{HD} = \{\ast\}$, $P_{HD}$ is the finite set of productions, and $S_{HD} = \{\text{module\_packet}\}$ is the initial graph.

$\text{PGGHD}$ is a context-free graph grammar that includes 70 productions and 888 attribute rules. Node labels [ ] and "" denote a nonterminal and terminal node, respectively. The node with the [module\_packet] label is the initial nonterminal. $\text{PGGHD}$ generates a directed graph that indicates a Hichart diagram. However the Hichart diagram is drawn in an undirected graph on a Hichart processing system for the sake of visibility. We call a laterally connected relation a parent-child relation and a longitudinally connected relation a sibling relation. The node to the left side of the current node is a parent node, the node to the right side is a child node, an upper node is an older brother node and a lower node is a younger brother. Hichart can implicitly distinguish each relation from the positional relation of nodes. Therefore $\text{PGGHD}$ omits edge labels. $\text{PGGHD}$ rewrites edge labels from empty labels to empty labels.

Figure 5 shows an example of the productions and semantic rules of $\text{PGGHD}$. The large rectangle labeled [\text{explanation\_module\_algorithm}] is a nonterminal node. A rewriting step of this production consists of removing a node labeled [\text{explanation\_module\_algorithm}] from a given host graph and substituting the graph consisting of [\text{explanation}] and [\text{module\_algorithm}].

Each production has semantic rules. The semantic rules compute the attributes for drawing a diagram, such as the coordinates, and generating SVG files for a given Hichart diagram.

![Fig. 4 Example of an Hichart for DXL.](image-url)

**Fig. 4 Example of an Hichart for DXL.**
4.2 Derivation of PGGHD

Let \( H \) be a given host graph and \( p' : X' \rightarrow (D', C') \) be a production copy of \( p : X \rightarrow (D, C) \in P \), where \( H \) and \( D \) are disjoint.

We substitute \((D, C)\) for a node in \( H \) as follows.

1. Remove a mother node \( X' \) and edges that connect \( X' \) from host graph \( H \).
2. Embed the daughter graph \( D' \) into \( H' \), and
3. Put edges between the nodes of \( D' \) and the nodes that were connected to the mother node in the \( H \) of \( H' \) by using the connection instructions of \( C' \).

We use the definition of the substitution in [6].

Figure 6 illustrates a derivation from the initial nonterminal labeled \([\text{module_packet}]\).

4.3 Precedence Relation for PGGHD

The existing graph grammar for Hichart/DXL has no theoretical guarantee that it can construct an effective parser. Below, we describe how we changed the existing graph grammar into a precedence graph grammar by referring to Kaul's precedence graph grammar [15]. We also prove and describe an example of calculating a precedence relation.

4.3.1 Modification of Previous Graph Grammar

The existing Hichart/DXL graph grammar has precedence conflicts. Figure 7 shows an example of such a precedence conflict.

Let \( G_a \) be a subgraph of a graph generated by the previous graph grammar. A production copy of production 58 in Fig. 7 is applied to node \( n_{10} \) of graph \( G_a \), from which graph \( G_b \) is obtained. Similarly, \( G_c \) is obtained after a production copy of production 60 in Fig. 7 is applied to \( n_{13} \) of \( G_b \). Since the \( n_{13} \) generated by the derivation specification \( s_b \) appears on the left hand side of the derivation specification \( s_c \), \( s_b \leq s_c \).

Let us compute the precedence relation between nodes \( n_{11} \) and \( n_{12} \). The derivation specifications that generated each node are \( S_D(n_{11}) = s_b \), \( S_D(n_{12}) = s_h \). \( n_{11} \) and \( n_{12} \) are generated by the same derivation specification \( s_b \), so the precedence relation between \( n_{11} \) and \( n_{12} \) is \( n_{11} \leq n_{12} \). The precedence relation between node label is \[ \{("if", *, \emptyset, [\text{branch_statement_list, cushion}]\}) \in R_e \].

Next let us compute the precedence relation between \( n_{11} \) and \( n_{14} \). Firstly, derivation specifications for nodes \( n_{11} \) and \( n_{14} \) are \( S_D(n_{11}) = s_h \) and \( S_D(n_{14}) = s_c \), respectively.

This implies the \( n_{11} \) and \( n_{14} \) are respectively generated by the derivation specifications \( s_h \) and \( s_c \). Therefore, it holds that \( s_D(n_{11}) \leq_D s_D(n_{14}) \), that is, \( n_{11} \leq n_{14} \). The precedence relation between nodes label \( n_{11} \) and \( n_{14} \) is \[ \{("if", *, \emptyset, [\text{branch_} \] \).
statement_list_cushion]) ∈ R_c. Consequently, a tuple ("if", ∗, ∅, [branch_statement_list_cushion]) exists in two precedence relations, which means there is a precedence conflict.

No precedence conflict is required to be a precedence graph grammar. Therefore we changed the existing graph grammar so as to have no precedence conflict.

We changed Hichart structure so that parent nodes at most one child node. Consequently, the generated graph represents the inner structure of Hichart. For example, we changed Gc of Fig. 7 to Gc of Fig. 8. Whereas the node n₁₁ with “if then” label in Fig. 7 has two child nodes, the node n₁₁ with “if then” label in Fig. 8 has one child.

4.3.2 Properties of PGGHD

In this section, we describe some properties of PGGHD. First, we prove the confluent and symmetric properties. After that, we prove that PGGHD is a precedence graph grammar.

Lemma 1: The grammar PGGHD is confluent.

Proof. Let Gₓ be a sentential form of the grammar PGGHD, vₓ and vᵧ be nonterminal nodes of Gₓ, and pₓ and pᵧ be productions that can apply to vₓ and vᵧ. Here we consider two derivation sequences Gₓ → Gₓ₁ → Gₓ₂, and Gₓ → Gₓ₁ → Gₓ₂. To prove the confluence of PGGHD, we have only to show that Gₓ₁ and Gₓ₂ are isomorphic.

First, consider the case in which there is no edge between vₓ and vᵧ of Gₓ. If there is no edge between vₓ and vᵧ, the nodes do not affect each other. Hence, it is trivial that PGGHD generates an isomorphic graph regardless of the order in which the productions are applied.

Next consider the case in which there is an edge between vₓ and vᵧ of Gₓ. All connection patterns generated by PGGHD are shown in Fig. 9. Black dots denote terminal nodes, and boxes denotes nonterminal nodes. The case in which there is an edge between vₓ and vᵧ of Gₓ occurs only when the nodes have type IV in Fig. 9 (nodes have a brother relation). Therefore, we have only to consider the case in which nodes have a brother relation. In connection type IV of Fig. 9, we can let vₓ be an upper node and vᵧ be a lower node without loss of generality. We control the node-labeling of the productions so that only type I, II, and III in Fig. 9 can apply vₓ, and only type I, II, III and IV in Fig. 9 apply vᵧ. As shown in Fig. 9, every production of PGGHD inherits the connections of the mother node. Thus when the productions pₓ and pᵧ are applied in any order of application, the connections between vₓ and vᵧ are reserved and their resultant graphs are isomorphic. Therefore, when the productions pₓ and pᵧ are applied, PGGHD generates isomorphic graphs independent of the application. Hence, Gₓ₁ and Gₓ₂ obtained by Gₓ → Gₓ₁ → Gₓ₂, and Gₓ → Gₓ₁ → Gₓ₂ are isomorphic. The above shows that PGGHD is confluent.

Figure 10 shows an example of confluence. Let Gₓ be a graph derived by PGGHD, and sₓ = (pₓ, n₁₁, Dₛ, bₛ), sᵧ =
Fig. 10 Example of confluence.

Fig. 11 Example of a node that is connected to two nonterminal nodes.

$(p_5, n_2, \tilde{D}_6, \tilde{b}_0)$ be derivation specifications. Figure 10 illustrates two different derivation sequences $G_5 \rightarrow G_6 \rightarrow G_8$ and $G_5 \rightarrow G_7 \rightarrow G_8$. The two derivation sequences generate an isomorphic graph regardless of the order in which the productions are applied.

**Lemma 2:** The grammar PGGHD is symmetric.

**Proof.** Figure 9 shows the four production types of PGGHD. All four types have only identity mappings as automorphisms. Therefore, $I_v = I_v$ holds for every production of PGGHD. Thus, all productions of PGGHD are symmetric, and therefore PGGHD is symmetric. \(\Box\)

**Lemma 3:** The grammar PGGHD has no precedence conflicts.

**Proof.** PGGHD generates a tree structure from the root node to the leaf node. Productions of Type I, Type II and Type IV in Fig. 9 can be applied to the bottom node of a Hichart structure. Let the bottom node be a node with no child and no younger brother. Moreover PGGHD has no reflexive nonterminal property.

PGGHD does not generate a node that is connected to nodes with same node label. In Fig. 11, the node label nonterminal A is always different from nonterminal B in PGGHD.

Suppose we obtain $G_{aBC}$ by deriving from $G_A$ in Fig. 12. If we firstly generate a new node in the horizontal direction, we can obtain only $G_{AB}$ type graph. In $G_{aB}$, the parent of the node [B] can not be a nonterminal node in PGGHD. So, PGGHD can not generate a new node in the vertical direction to obtain $G_{aBC}$ in the case of $G_{aB}$. In order to generate $G_{aBC}$, it is necessary to generate $G_{AC}$ first by applying production of Type IV. Therefore only the derivation $G_A \rightarrow G_{AC} \rightarrow G_{aBC}$ can generate $G_{aBC}$ graph. Thus PGGHD has a restriction of derivation patterns and labeling pair of node labels. Therefore a precedence conflict does not occur in PGGHD. \(\Box\)

**Theorem 1:** The grammar PGGHD is a precedence graph grammar.

**Proof.** There are no two productions that have the same label tuple on the right-hand-side of the productions. Therefore, the grammar PGGHD is uniquely invertible.

There is no nonterminal node such that it can be derived from another nonterminal node with the same node label in at least one step. Hence, there is no reflexive nonterminal node in PGGHD.

From Lemma 1, Lemma 2, Lemma 3 and the above, PGGHD is a precedence graph grammar. \(\Box\)

4.3.3 Example of Calculating Precedence Relations

Precedence relations are computed by a derivation sequence and by referencing derivation specifications. A derivation sequence is a sequence of derivation step, and derivation specifications give detailed information such as the applied production and mother node.

First, we compute the derivation sequence for a given Hichart diagram. Next, we compute relations between each derivation specification of the derivation sequence by using the obtained transitive closure of the derivation specifications. After that, we calculate precedence between node of a graph generated by PGGHD from the relation for each derivation specification. There are four types of precedence relation between nodes: \(\preceq\), \(<\), \(\succ\) and \(\gg\) . Finally, we compute precedence relations between label tuples by using the nodes precedence relations. Precedence relations are calculated in this manner between all nodes of the sentential form for PGGHD.

We show an example precedence relation calculation for Fig. 6. This example shows the result after applying four productions to the initial nonterminal with the [module_packet] label. The derivation sequence of the example
is \( G_0 \rightarrow G_1 \rightarrow G_2 \rightarrow G_3 \rightarrow G_4 \).

Now, we add precedence relations to PGGHD. Using Definition 8, the derivation specifications for Fig.6 are as follows.

\[
s_1 = (P_1, \text{[module]_packet}, \bar{D}_1, \bar{b}_1) \\
\begin{align*}
s_2 &= (P_3, \text{[profile]_module_list}, \bar{D}_2, \bar{b}_2) \\
s_3 &= (P_4, \text{[profile]}, \bar{D}_3, \bar{b}_3) \\
s_4 &= (P_5, \text{[module]_list}, \bar{D}_4, \bar{b}_4)
\end{align*}
\]

The node 3 with the [profile]_module list label is a \( L_2 \) of \( s_2 \) and is an induced subgraph of \( R_1 \) of \( s_1 \). \( s_1 \) precedes \( s_2 \), that is \( s_1 \leq_D s_2 \). Similarly, \( s_1 \leq_D s_2 \leq_D s_3 \) and \( s_1 \leq_D s_2 \leq_D s_4 \), or more precisely \( s_1 = \{(s_1, s_2), (s_2, s_3), (s_1, s_3), (s_2, s_4), (s_1, s_4)\} \).

The way to compute a precedence relation between nodes 6 with the "profile" label and 8 with the [module] label is as follows. \( s_3 \Rightarrow s_4 \) because \( s_4(6) = s_3, s_4(8) = s_4 \). Therefore, \( s_4(6) \gg s_4(8) \). The precedence relation between labels is \( \text{lab}_G(6, 8) = (\text{"profile"}, *, \emptyset, \text{[module]} ) \in R_x \).

The precedence relation between the node "profile" generated by \( p_4 \) and [module] generated by \( p_5 \) is incomparable because the derivation specifications \( s_3, s_4 \) are neither reflexive nor transitive closure.

We defined all the precedence relations for PGGHD. The details of the grammar and precedence relation table for PGGHD are described in [18].

5. Parsing of Graph Grammar for Hichart/DXL

5.1 Parsing Algorithm for PGGHD

This section describes the parsing algorithm for PGGHD. This parser uses a stack for storing traversed nodes, and it starts parsing from the root node of the input graph.

First, we review the shift and reduce operations [15].

An instantaneous description is \( (G, K, \Psi) \), where \( G \) is the instantaneous graph, \( K \) is an ordered list of nodes in \( G \), and \( \Psi \) is a set of derivation specifications. Let \( (G, K, \Psi) \) be an instantaneous description, \( K = < v_1, \ldots, v_k >, k \geq 1 \). Let \( j \) be the minimum index \( 1 \leq j \leq k \) such that there is some path in \( G \) from \( v_1 \) to \( v_j \) along equal precedence. \( \text{TOP}(G, K) \) is defined as \( G \mid \{v_j, \ldots, v_k\} \).

\[(G, K, \Psi) \vdash_s (G, Kw, \Psi) \] is a shift if \( (i) \ v \in V_G \) does not occur in \( K \) and \( (ii) \) \( \text{lab}(v, w) \in R_e \cup R_r \) for some \( v \) in \( \text{TOP}(G, K) \).

\[(G, K_1 K_2, \Psi) \vdash_R (G', K_1 w, \Psi \cup \{s\}) \] is a reduce if \( (i) \) a node \( w \) is not used in \( G \) or \( \Psi \), \( (ii) \) \( s = (p, \bar{L}, \bar{R}, \bar{b}) \), \( G' \rightarrow G \) is a derivation step, \( s \notin \Psi \), \( K_2 = V_R \), \( G \mid K_2 = \text{TOP}(G, K) \) is a precedence handle, and \( G' \mid \{w\} = \bar{L}, K_1 K_2 \) denotes the concatenation of both sequences \( K_1 \) and \( K_2 \).

The parsing algorithm for PGGHD is as follows:

The input for Algorithm 1 (PGGHD_Parser) is a graph that indicates the inner structure for the Hichart diagram. PGGHD_Parser repeats Procedure 1 (Precedence_Analysis) until host graph becomes the initial graph or a syntax error is detected. If the input graph can be parsed, its parse tree is generated by line 6 to 8 of PGGHD_Parser.

In Precedence_Analysis, the parser gets all nodes from \( K \) such that the nodes have the same precedence as the Top and there is no node with a different precedence between the Top and them at line 2. In loop lines 6 to 14, the parser tries to find nodes that can perform a shift of child nodes or younger brother nodes. Note that this loop is executed at most two times per executing the procedure Precedence_Analysis.

The shift is executed if the current node has a child or younger brother and has higher or equal precedence com-
pared to it. If there is no shift node, the parser does lines 15 to 22. If there is no shift node, the parser generates a graph structure from nodeList as a precedence handle. The parser searches for a production where the right-hand-side is isomorphic to the precedence handle. If the parser finds such a production, it updates graph and shift_Nodes with the production copy.

5.2 Example of Parsing

In Fig. 13, (a) describes an input graph H and shifted-Nodes K is an ordered list of H. K is a stack that stores information for the shifted nodes. The parser sets the root node of the input graph to be the current node and stores the root node information on stack K.

The parser operates as follows. First, the parser computes the TOP(H, K), which is a list of nodes that has an equal precedence relation between node labels from the top of stack K. In (a), the result for TOP(H, K) has only one node with the label "m_packet". If the current node with "m_packet" label has an ascending precedence between it and its child node, a shift is performed; that is, "profile" is stored in K. Subsequent shifts are repeated in a similar manner.

In Fig. 13 (c), a precedence handle is found since there is no higher node. The parser then searches for a production in which the right-hand-side is isomorphic to the precedence handle. In this case, the parser finds Production 10. The parser then reduces the precedence handle to the left-hand-side of the production. Figure 13 (d) illustrates the situation after the graph has been reduced.

The above operations are repeated until the graph becomes the initial graph with the [module_packet] label. Otherwise the parser detects a syntax error.

5.3 Complexity of the Parsing Algorithm

In this section, we show that the time complexity of PG-GHD_Parser is linear with respect to the number of nodes in an input graph.

**Theorem 2:** PG-GHD_Parser executes in \(O(n)\) where \(n\) is the number of nodes in an input graph.

**Proof.** First, we investigate the complexity of Precedence_Analysis.

The procedure getTop at line 2 of Precedence_Analysis can be computed in \(O(1)\) because the right-hand-side of PG-GHD has two nodes at most and getTop can determine a handle by popping one or two nodes from the stack K.

The for loop between lines 6 and 14 executes in \(O(1)\). At most two nodes are obtained by getTop, so the for loop between lines 6 and 14 is repeated two times. The shift procedure at lines 8 or line 11 shifts the child node or younger brother node once. Thus, the shift procedure executes shift in \(O(1)\) time.

getHandle at line 15 can be computed in \(O(1)\). If there is one node in the nodeList, a handle is computed in constant time. If there are two nodes in the nodeList, a handle is computed by checking the parent-child and brother relations of (at most two) nodes in the nodeList. Therefore getHandle executes in \(O(1)\).

The complexity of findProduction at line 16 is \(O(1)\). findProduction compares a handle with productions of PG-GHD. The order of the handle is at most 2 and that of the right-hand-side of the production is also at most 2. Moreover, the number of productions is 70. Hence, findProduction executes in \(O(1)\).
The complexity of ReplaceGraph at line 18 is \(O(1)\). ReplaceGraph replaces a handle with the left-hand-side of a production and makes it connect to the rest graph. Since the maximum degree of graphs generated by PGGHD is 4, this process can be done in \(O(1)\). Therefore, the complexity of ReplaceGraph is \(O(1)\).

The above shows that Precedence Analysis executes in \(O(1)\).

Next, we investigate the complexity of the PGGHD_Parser algorithm.

The while loop between lines 3 and line 5 executes \(8n\) times. This parser reduces one node by executing Precedence Analysis at most eight times. Hence, PGGHD_Parser only executes Precedence Analysis at most \(8n\) times for a graph with \(n\) nodes.

On lines 6-8, the parse tree is generated by traversing the derivation sequence obtained from the results of executing Precedence Analysis in the reverse manner once.

Therefore, the complexity of PGGHD_Parser is \(O(n)\) where \(n\) is the number of nodes in the input graph.

6. Hichart Editor

We developed a graphical editor based on precedence graph grammar for Hichart/DXL (PGGHD). The graphical editor has features for parsing, aesthetic drawing, and generating SVG files for Hichart/DXL diagrams. The SVG files are generated by evaluating attributes for SVG. The graphical editor consists of about 10000 lines of Java.

6.1 Features of Hichart Editor

The Hichart editor is a graphical editor supporting structure-free or free-hand editing. Users can directly manipulate a diagram with this editor, so that the generated diagram can be analyzed by a parser based on the graph grammar. Hichart diagrams are input into the editor. The editor outputs a Hichart code with a derivation tree and SVG files with aesthetically drawn Hichart diagrams.

Figure 14 illustrates an example of Hichart editor screens for free-hand editing. Figure 15 shows a screenshot after the diagram of Fig. 14 has been automatically drawn on it.

The main features of the Hichart editor are: (1) it checks the correctness of Hichart diagrams by using the parser, (2) it draws Hichart diagrams aesthetically, (3) it generates an SVG file for a given program diagram, and (4) ensures that the display for the SVG file on a general Web browser directly corresponds to the diagram in the editor.

6.2 Method of Implementing Features

In this section, we describe the methods by which we implement the features based on attribute graph grammars. Attributes allow us to describe a process explicitly. They also have modularity with respect to program descriptions and are written in a declarative manner. Therefore, system developers can easily maintain and write their own attributes.

6.2.1 Drawing Aesthetic Diagrams

Attributes of PGGHD such as coordinates are computed by calculating semantic rules on the derivation tree. The number of semantic rules concerning coordinates is 781.

When users execute an editor command, nicely drawn diagrams can be automatically drawn by evaluating layout attributes. The evaluation is executed by traversing on the derivation tree. Figure 15 shows the screenshot after the
We introduce an attribute $S_{SVG}$.

6.2.2 Generation of SVG Documents with Aesthetic Drawings

We introduce an attribute $S_{SVG}$, which contains SVG source codes, because its value and representation correspond to the Hichart diagram. Then, we define semantic rules to satisfy the constraints of drawing aesthetic Hichart diagrams. The number of semantic rules concerning SVG is 107.

The SVG source codes are generated by evaluating $S_{SVG}$. Figure 16 illustrates the flow of generating SVG files. The attribute evaluation is performed in a bottom-up manner on the derivation tree. Figure 17 is an example of the display of a Hichart diagram in SVG.

7. Discussion

We constructed a graph grammar for the program diagram Hichart. We have been researching program diagrams for twenty years. Lately, we have been studying tabular diagrams called Hiform for program specification forms. We modeled a program specification table structure using a graph and defined its graph grammar. We formalized the program diagram Hichart based on graph grammar in order to prepare a unified program diagram and program specification.

Our graph grammar for Hichart generates a tree-structure graph. There have been some studies on tree grammars, [19]–[21]. Simpler tree grammars are sufficient for generating Hichart diagrams. The application of tree grammars to Hichart will be dealt with in the future.

8. Conclusion

We described the graph grammar PGGHD for Hichart/DXL and proved that it is a precedence grammar. Moreover, we designed and implemented a linear time parser for PGGHD by using its precedence and constructed a processing system for PGGHD.

Our attribute graph grammar approach is applicable to other visual programming systems that handle tree-like graphs. In the future, we will extend our approach to other languages such as object-oriented language.

Acknowledgments

This research was partially supported by the INOUE ENRYO Memorial Foundation for Promoting Sciences. The authors thank the reviewers for their valuable comments.

References

[18] PGGHD (Precedence Graph Grammar for Hichart/DXL) Web Site: http://www.w3.org/TR/SVG/


Takaaki Goto received an M.S. degree from Toyo University in 2003. He is currently a graduate student in Toyo University. His main research interests are applications of graph grammars, visual languages, and software development environments.

Kenji Ruise received his M.S. degree from Nihon University in 2001. He has been a school teacher at Kirigaoka School for the Physically Challenged University of Tsukuba since 2001. His main research interests are the applications of graph grammars, visual languages, software development environments.

Takeo Yaku graduated from Jiyu Gakuen College in 1970. He received MSc and DSc from Waseda University in 1972 and 1977, respectively. He has been a Professor at the Department of Computer Science and System Analysis of Nihon University since 1992. His research interests include software visualization, human interface, graph languages, and graph algorithms. He is a member of the IEEE Computer Society and ACM.

Kensei Tsuchida received M.S. and D.S. degrees in mathematics from Waseda University in 1984 and 1994 respectively. He was a member of the Software Engineering Development Laboratory, NEC Corporation in 1984-1990. From 1990 to 1992, he was a Research Associate of the Department of Industrial Engineering and Management at Kanagawa University. In 1992 he joined Toyo University, where he was an Instructor until 1995 and an associate professor from 1995 to 2002, and since 2002 he has been a Professor of the Department of Information and Computer Sciences. He was a visiting associate professor of the Department of Computer Science at Oregon State University from 1997 to 1998.