情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-MPS-84 No.1
2011/7/18

UMLパッケージ図に対するグラフ文法とその応用

後藤隆彰[†1]　西野哲朗[†2]　土田賢省[†3]

図式表現は，その視認性の良さからソフトウェア設計や開発においてよく利用されている．一方，近年 UML(Unified Modeling Language) が提案され，これまでに多くのシステムの分析や設計，実装に用いられている．

これら図式表現をコンピュータ上で自動的に処理するためには，まずプログラム図の構文を定義する必要がある．また，プログラム図などの 2 次元データを対象として構文解析を行うためには，各要素間の関係が記述される必要がある．本研究では，UML のパッケージ図の生成を，グラフ文法に基づいて実現し，図式の自動処理を行うための枠組みを提案する．

An Attribute Graph Grammar for UML Package Diagrams and its Applications

Takaaki Goto,[†1] Tetsuro Nishino[†2]
and Kensei Tsuchida[†3]

Graphical representations are often used in software design and development because of their expressiveness. Unified Modeling Language (UML) for modeling in software development was proposed recently, and in 2005 it was standardized as the ISO/IEC 19501 standard.

In order to automate processing of these graphical representations using computers, a syntax for program diagrams must first be defined. We propose a framework for specifying these diagrams using a graph grammar, and for processing these diagrams automatically.

†1 電気通信大学 産学官連携センター
Center for Industrial and Governmental Relations, The University of Electro-Communications
†2 電気通信大学 情報理工学研究科
Graduate School of Informatics and Engineering, The University of Electro-Communications
†3 東洋大学 総合情報学部
Faculty of Information Science and Arts, Toyo University

1. Introduction

Graphical representations are often used in software design and development because of their expressiveness. Various graphical program description languages have been reported, including Hierarchical flowchart language (Hichart), Problem Analysis Diagrams (PAD), Hierarchical and Compact description charts (HCP), and Structured Programming Diagrams (SPD), and many Computer Aided Software Engineering (CASE) tools have been developed based on these languages[1)–3)].

On the other hand, the Unified Modeling Language (UML) for modeling in software development was proposed recently compared with above graphical program description languages, and in 2005 it was standardized as the ISO/IEC 19501 standard. UML has already been used in the analysis, design and implementation of many systems. It makes use of various types of diagrams, such as class and sequence diagrams, for designing processes in system development, from upstream process to downstream process. In order to automate processing of these graphical representations using computers, a syntax for program diagrams must first be defined. Then, in order to analyze the syntax of two-dimensional objects such as program diagrams, the relationships between each of the elements must also be described. Graph grammars are one possible effective means for implementing these methods. Graph grammars provide a formal method that enables rigorous definition of mechanisms for generating and analyzing graphs.

Research on graph grammars has been done by Rozenberg[4)] and others. Research has also been done on UML[5)] and graph grammars and graph transformations with respect to UML[6)–8)].

However these researches do not deal with syntax formalization for visual representation. And also graph grammars for package diagram are not proposed yet in previous researches. Therefore we provide a graph grammar for package diagram of UML to propose theoretical fundamentals of UML.

With regard to Web documents, XML and SVG have been proposed as standard document and graphical formats for the Web. Scalable Vector Graphics (SVG)[9)] 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 avail-

able Web browser. With these formats, users can share document 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.

The goal of this research is to generate UML package diagrams based on a graph grammar. We propose a framework for specifying these diagrams using a graph grammar, and for processing these diagrams automatically.

2. Preliminary

2.1 Graph Grammars

Definition 1. ([4]) An *edNCE graph grammar* is a six-tuple $GG = (\Sigma, \Delta, \Gamma, \Omega, P, S)$, where Σ is the alphabet of node labels, $\Delta \subseteq \Sigma$ is the alphabet of terminal node labels, Γ 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 $X \to (D, C)$ where X is a nonterminal node label, D is a graph over Σ and Γ, and $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_D \times \{in, out\}$ is the connection relation which is a set of connection instructions. A pair (D, C) is a graph with embedding over Σ and Γ. □

An example of a production is shown in Figure 1. In the Figure, a box is a nonterminal node and a filled circle is a terminal node. X, Y, and b mean node labels and v_0, v_1, and v_2 mean node IDs. Nodes with same node label can appear in a graph, while nodes with same node ID will never appeared in a graph. The production of Figure 1 indicates that after the removal of a nonterminal node with label X, embed the graph consists of terminal node with label b and the nonterminal node with label Y. Each production has connection instructions. The connection instruction of this production is $(a, \alpha/\beta, v_1, in)$, however this connection instruction is not described in the notation of Figure 1.

In Figure 2, the production of Figure 1 and its connection instruction are drawn simultaneously. The large box of Figure 2 indicates the left-hand side, and two nodes with label b and Y are right-hand side of the production of Figure 1.

An example of application of the production is shown in Figure 3. In Figure 3 $H = (V_H, E_H, \lambda_H)$ is a graph with $V_H = \{n_1, n_2\}$, $E_H = \{(n_1, \alpha, n_2)\}$, $\lambda_H(n_1) = a$,

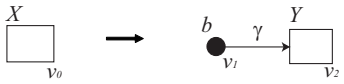

Fig.1 An example of a production

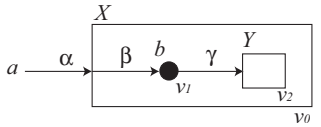

Fig.2 An example of a production with the connection relation

and $\lambda_H(n_2) = X$. The production copy p' of p is as follows: $p' : X \to (D', C')$ where $X = \lambda_H(n_2)$, $D' = (V_{D'}, E_{D'}, \lambda_{D'})$ such that $V_{D'} = \{n_3, n_4\}$, $E_{D'} = \{(n_3, \gamma, n_4)\}$, $\lambda_{D'}(n_3) = b$, $\lambda_{D'}(n_4) = Y$ and $C' = \{(a, \alpha/\beta, n_3, in)\}$.

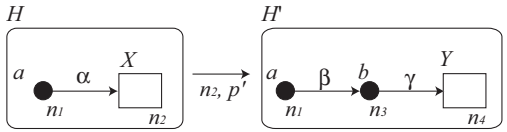

Fig.3 An example of applying a production rule

In Figure 3, H indicates the host graph and H' is the resulting graph. At first, we remove the node X and edges that connect with node X from host graph H. Next we embed the daughter graph, including node b and node Y. Then we establish edges between the nodes of daughter graph and the nodes that were connected to the node X using the connection instructions on the production p'. Therefore the edge label α is rewritten to β by the production p'.

Definition 2. ([10),11]) An *Attribute edNCE Graph Grammar* is a six-tuple $AGG = \langle GG, Att, F \rangle$, 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 $X \to (D, C)$.

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 $Att(X) = Inh(X) \cup Syn(X)$. $Att = \bigcup_{X \in \Sigma} Att(X)$ 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 $Syn(X_0) \bigcup_{X \in Lab(D)} Inh(X)$. A semantic rule defining an attribute $a_0(X_{i_0})$ has the form $a_0(X_{i_0}) := f(a_1(X_{i_1}), \cdots, a_m(X_{i_m}))$. Here f is a mapping from $V(a_1(X_{i_1})) \times \cdots \times V(a_m(X_{i_m}))$ into $V(a_0(X_{i_0}))$. In this situation, we say that $a_0(X_{i_0})$ depends on $a_j(X_{i_j})$ 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. □

Attribute values are calculated by evaluating attributes according to semantic rules on the derivation tree.

2.2 UML

Unified Modeling Language (UML) is a notation for modeling object oriented system development using diagrams. UML can be divided into structural diagrams and behavioral diagrams. Structural diagrams are used to describe the structure of what is being modeled and include class, object, and package diagrams, and so on. Behavioral diagrams are used to describe the behavior of what is being modeled and include such as use-case, activity, and state-machine diagrams.

Structure diagrams include class diagrams, which describe the static relationships between classes, and package diagrams, which group classes and describe relationships between packages and package nesting relationships.

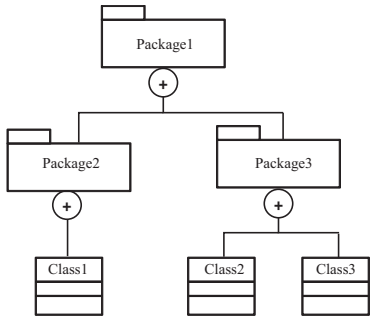

Fig.4 An example of a package diagram

Figure 4 shows an example of a package diagram. The box with rectangle at the upper left indicates a package. The box with three compartments is a class. Each of three parts indicates its class name, its attribute, and its methods from top to the bottom. A plus with circle is used to represent which components the package contains. Package 1 contains Package 2 and Package3, and Package 3 contains Class2 and Class 3.

3. Graph Grammar for UML Package Diagrams

In this section we describe our Graph Grammar for Package Diagrams (GGPD), for UML package diagrams.

3.1 Grammar Overview

Definition 3. The Graph Grammar for Package Diagrams (GGPD), for UML package diagrams, is a six-tuple $GGPD = (\Sigma_{PD}, \Delta_{PD}, \Gamma_{PD}, \Omega_{PD}, P_{PD}, S_{PD})$. Here, Σ_{PD} = { S, A, T, L, R, M, rop, sp, lep, rip, mip, lec, mic, ric } is a finite set of node labels, Δ_{PD} = { rop, sp, lep, rip, mip, lec, mic, ric } is a finite set of terminal node labels, Γ_{PD} = { $*$ }, Ω_{PD} = { $*$ }, P_{PD} = { $P_1, ..., P_{17}$ } is a finite set of production rules, and S_{PD} = { S }, is the initial non-terminal. □

The GGPD generates package hierarchy diagrams. It is a context-free grammar and there are 17 production rules. An example of GGPD production rule is shown in Figure 5.

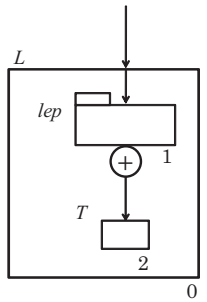

Fig.5 An Example of a production rule of GGPD

In the figure, the production rule can be applied to a node labeled L, which is a non-terminal node, to generate a terminal node with the label lep, representing a package, and a non-terminal node labeled T.

A node with capitalized label indicates a nonterminal node, and a node with uncapitalized label indicates a terminal node. Our grammar generates directed graphs. However obtained graphs are drawn without arrows by assumption that the direction of each edge from top down.

3.2 Example of Derivation

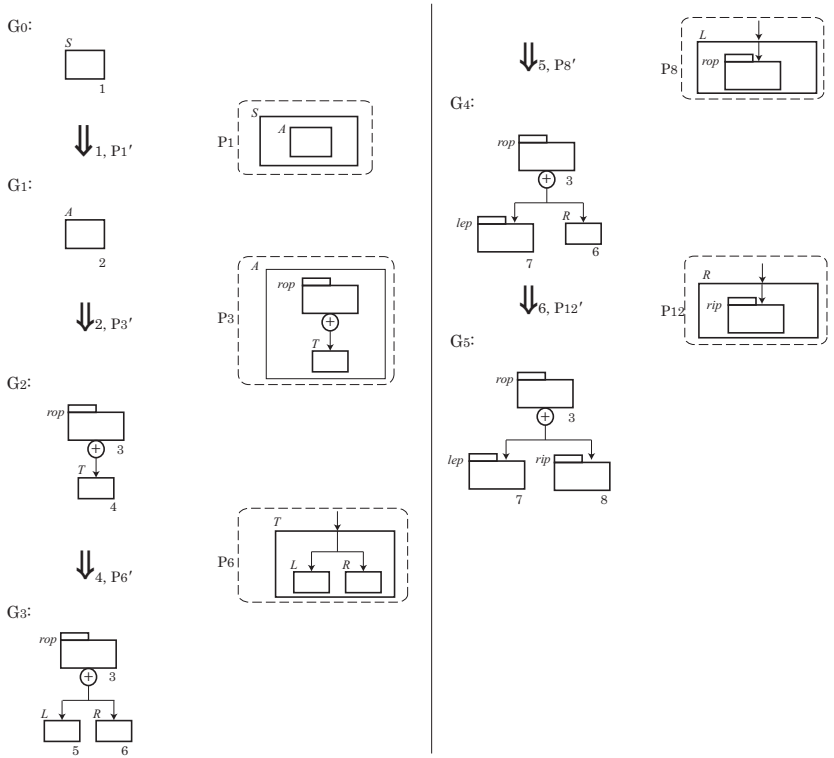

Fig.6 An example of a GGPD derivation

Figure 6 shows an example of a GGPD derivation. In this example, G_0 is a graph with the node labeled S. The node ID is 1 (lower right of the node).

Then the production rule P_1 is applied to a non-terminal node labeled S with node ID 1, which is the initial non-terminal node. That is, remove a mother node with label S and node ID 1, then embed a daughter graph in the P_1. In this case the daughter graph is the node with label A. This produces the non-terminal node labeled A with node ID 2, to which the P_3 production rule is applied. That is, graph G_1 consists of node with node ID 2 is obtained.

After application of the production P_3, the terminal node labeled *rop* and a non-terminal node labeled T are generated. We apply productions to obtain a graph that correspond to UML package diagrams.

We can obtain a derivation tree from derivation sequence of production. Figure 7 shows the derivation tree corresponding to Figure 6. In the Figure 7, the labels show the name of production rules.

The sequence of production rules applied can be expressed in terms of a production rule derivation tree.

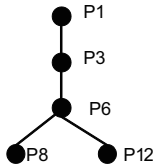

Fig.7 A derivation tree corresponding to the tree in Figure 6

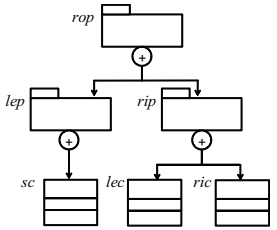

Fig.8 An example of package diagram resulting from derivation

Another example of a package diagram resulting from applying the production rules is shown in Figure 8.

3.3 Generation of SVG document for package diagrams

We introduce attribute S_{SVG} which contains SVG source codes, as its value and representation corresponding to the package diagram. We have a plan to generate diagrams with animation. SVG can display on browser such as IE with SVG plugin.

SVG source codes are generated by evaluating S_{SVG}. Evaluation of attributes is performed in the bottom-up manner on derivation trees. Figure 9 illustrates the flow of

generating SVG files.

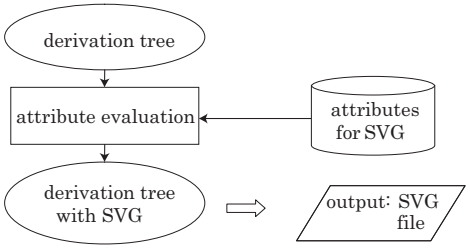

Fig.9 Flow of generating SVG files

Figure 10 gives examples of semantic rules with the attribute S_{SVG}.

```
Ssvg(1)=<rect x="x(1)" y="y(1)" width="w(1)" height="h(1)"
        fill="white" fill-opacity="1" stroke="black" />...
Ssvg(0)=Ssvg(1)Ssvg(2)
```

Fig.10 An examples of semantic rules with the attribute S_{SVG}

3.4 Folding / UnFolding

When drawing package diagrams for large-scale systems, the scale of diagrams can become large, and this can make diagrams difficult to comprehend visually. This makes it necessary to process diagrams to summarize and hide information. Thus, we perform information-hiding by expressing diagrams in sentential form.

Figure 11 shows an example of a package diagram and its derivation tree before folding, and Figure 12 shows the package diagram and derivation tree after folding.

4. UML Package Diagram Editor

In this section, we explain our prototype UML package diagram editor based on the grammar described in Section 3. The editor is a syntax-directed editor and was developed in Java.

On the editor, when a non-terminal node displayed on the editor screen is selected , a screen displaying the production rules that can be applied to the non-terminal node is displayed. Figure 13, 14, and 15 show a screen shot when the non-terminal node with node ID of 2 and labeled A in the package diagram editor screen is clicked (Figure 13), and the applicable production rules are displayed (Figure 14). After choosing a production rule, the production rule is applied to the non-terminal node (Figure 15).

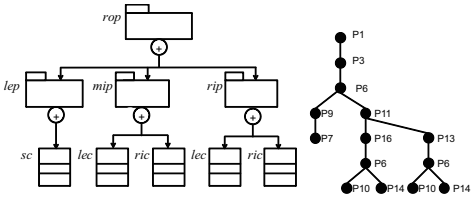

Fig.11 A package diagram and its derivation tree before folding

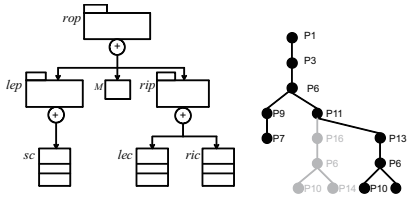

Fig.12 A package diagram and its derivation tree after folding

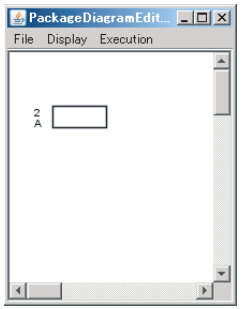

Fig.13 A nonterminal node on the editor

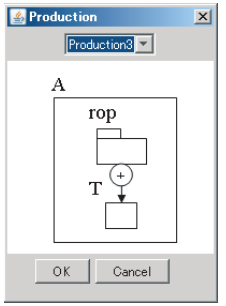

Fig.14 The production rule display screen of the package diagram editor

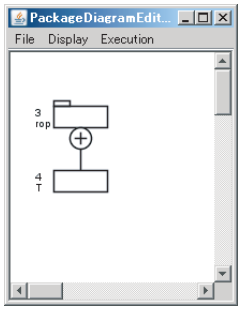

Fig.15 An example of applying production rule on the editor

The applied production rules can also be displayed as a derivation tree, as shown in Figure 16.

When users execute an editor command, SVG files can be automatically generated by evaluating SVG attributes. The evaluation is executed by traversing on the derivation tree. Figure 17 is an example of the display of a package diagram in SVG.

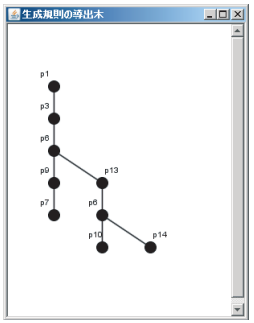

Fig.16 The derivation tree display screen of the package diagram editor

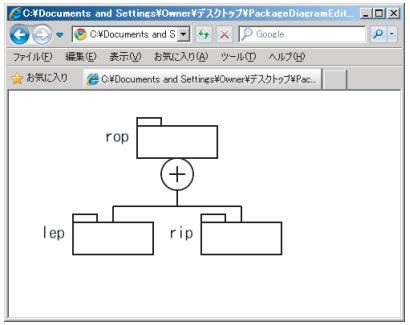

Fig.17 An example of the display of a package diagram in SVG

5. Conclusion

In this paper, we have defined a graph grammar for generating the hierarchical structure of UML package diagrams. We have also created a syntax-directed diagram editor for the defined grammar. A future issue for study is to implement syntactic analysis. The editor developed here is able to generate diagrams according to the grammar and complying with the syntax through human intervention, but it is not able to determine, from an arbitrary input, whether a diagram conforms or not. By implementing syntactic analysis, automatic processing of arbitrary input diagrams will be possible. Application of this technology to automatic generation of software documentation is an another possibility.

References

1) Harada, K.: *Structure Editor*, Kyoritsu Shuppan (1987). (in Japanese).
2) Yoshihiro Adachi, Youzou Miyadera, Kimio Sugita, Kensei Tsuchida and Takeo Yaku: A Visual Programming Environment Based on Graph Grammars and Tidy Graph Drawing, *Proceedings of The 20th International Conference on Software Engineering (ICSE '98)*, Vol.2, pp.74–79 (1998).
3) T. Goto, K. Ruise, T. Yaku and K. Tsuchida: Visual Software Development Environment Based on Graph Grammars, *IEICE Transactions on Information and Systems*, Vol.92, No.3, pp.401–412 (2009).
4) Rozenberg, G.: *Handbook of Graph Grammar and Computing by Graph Transformation Volume 1*, World Scientific Publishing (1997).
5) Kotulski, L. and Dymek, D.: On the Modeling Timing Behavior of the System with UML(VR), *Computational Science ICCS 2008*, Lecture Notes in Computer Science, Vol.5101, pp.386–395 (2008).
6) Hermann, F., Ehrig, H. and Taentzer, G.: A Typed Attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams, *Electron. Notes Theor. Comput. Sci.*, Vol.211, pp.261–269 (2008).
7) Kong, Jun and Zhang, Kang and Dong, Jing and Xu, Dianxiang: Specifying behavioral semantics of UML diagrams through graph transformations, *J. Syst. Softw.*, Vol.82, pp.292–306 (2009).
8) Petriu, D. and Shen, H.: Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications, *Computer Performance Evaluation: Modelling Techniques and Tools*, Lecture Notes in Computer Science, Vol.2324, pp.183–204 (2002).
9) W3C Web Site. Scalable Vector Graphics (SVG): http://www.w3.org/TR/SVG/.
10) T. Nishino: Attribute Graph Grammars with Applications to Hichart Program Chart Editors, *Advances in Software Science and Technology*, Vol. 1, pp. 89–104 (1989).
11) T. Arita, K. Sugita, K. Tsuchida, and T. Yaku: Syntactic Tabular Form Processing by Precedence Attribute Graph Grammars, *Proc. IASTED Applied Informatics 2001*, pp.637–642 (2001).