IPSJ SIG Technical Report

Vol.2012-MPS-89 No.9
2012/7/16

Automatic Generation of Diagram Explanation based on an Attribute Graph Grammar

Takaaki Goto[1,a) Tetsuro Nishino[1,b) Kensei Tsuchida[2,c)

Abstract: Unified Modeling Language (UML) has already been used in the analysis, design, and implementation of many systems. Open Source Software (OSS) is often used in software development. However, it is often the case that OSS does not contain adequate documents, so the generation of software documents is important. Documents with diagrams are especially important to understand software, so it is important to generate documents with diagrams. In order to process large-scale diagrams, or many source codes automatically, formal and declarative representation is needed. In this paper, we propose automatic generation of documentation based on an attribute graph grammar.

1. Introduction

In the software development environment, it is desirable to have features that support programming. Many effective tools have been developed to provide a framework for developing reliable programs. In software development, software documents are very important to develop or modify systems. Graphical representations such as flowchart or Unified Modeling Language (UML) are often used in software design and development because of their expressiveness.

UML for modeling in software development has been proposed recently. In 2005, ISO/TEC 19501 became the 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 to downstream processes. In order to automate the 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. The authors of the current paper have proposed a graph grammar for package diagrams of UML [1].

Open Source Software (OSS) is often used in software development. However, it is often the case that OSS does not contain adequate documents. Software documents are needed in order to use or modify OSS, however, there are so many examples of source code with no documents, therefore, we need to generate documents from source code. Moreover, documents with diagrams are especially important to understand software, so it is important to generate documents with diagrams. This is why we started our research so that we can obtain documents with diagrams automatically.

Thus far, much research has targeted UML or software documents. Research has also been done on UML [2] and graph grammars and graph transformations with respect to UML [3], [4], [5]. However, no research focuses on syntax formalization for visual representation. As for software document generation, some research has been done [6], [7], [8]. This research targets documentation generation through program source code or diagrams. The formal method is not treated in this framework.

In order to process large-scale diagrams, or large amounts of source code automatically, formal and declarative representation is needed. In this paper, we propose automatic generation of documentation based on an attribute graph grammar.

2. Preliminary

2.1 Graph Grammars

Definition 1. ([9],[10]) Let Σ be an alphabet of node labels and Γ be an alphabet of edge labels. A *graph* over Σ and Γ is a tuple $H = (V, E, \lambda)$, where V is the finite 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. $E(v, w) \overset{\text{def}}{=} \{\gamma \in \Gamma \mid (v, \gamma, w) \in E\}$. The *label tuple* of two nodes $v, w \in V$ is $lab(v, w) \overset{\text{def}}{=} (\lambda(v), E(v, w), E(w, v), \lambda(w))$. □

Definition 2. ([9]) A *graph with* (neighborhood controlled) *embedding* over Σ and Γ is a pair (H, C) with $H \in GR_{\Sigma,\Gamma}$ and $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_H \times$ {in, out}. C is the *connection relation* of (H, C), and each element $(\sigma, \beta, \gamma, x, d)$ of C (with $\sigma \in \Sigma, \beta, \gamma \in \Gamma, x \in V_H$, and $d \in$ {in, out}) is a *connection instruction* of (H, C). A connection instruction $(\sigma, \beta, \gamma, x, d)$ will always be written as

1 Graduate School of Informatics and Engineering, The University of Electro-Communications, Chofu, Tokyo 182-8585, Japan
2 Faculty of Information Science and Arts, Toyo University, Kawagoe, Saitama 350-8585, Japan
a) gototakaaki@uec.ac.jp
b) nishino@ice.uec.ac.jp
c) kensei@toyo.jp

$(\sigma, \beta/\gamma, x, d)$. 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 $C_K = \{(\sigma, \beta/\gamma, f(x), d) \mid (\sigma, \beta/\gamma, x, d) \in C_H\}$. The set of all graphs with embedding over Σ and Γ is denoted as $GRE_{\Sigma,\Gamma}$. □

Figure 1 shows an example of a graph. In Figure 1 $H = (V_H, E_H, \lambda_H)$ is a graph with $V_H = \{n_1, n_2\}$, $E_H = \{(n_1, \alpha, n_2)\}$, $\lambda_H(n_1) = a$ and $\lambda_H(n_2) = X$. Here, n_1 and n_2 indicate node ID. Furthermore a, and X indicate node labels, nodes with a lowercase node label and with a uppercase node label are terminal node and nonterminal node, respectively,

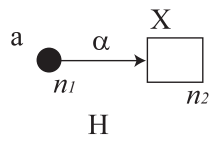

Fig. 1 An example of a graph

Definition 3. ([9]) 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. □

Definition 4. (cf. [9], [10]) 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 $p'_i : X \to (D'_i, C'_i) \in P$ 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 $p'_i \in copy(P)$, $\lambda_{H_{i-1}(v_i)} = X$, $D'_i \cong D$, $b'_i : V_{D'_i} \to V_{D_i}$.

We write $H_{i-1} \to_{v_i,p'_i} H_i$, or just $H_{i-1} \underset{s_i}{\to} H_i$, if $\lambda_{H_{i-1}}(v_i) = X$ and $H_i = H_{i-1}[v_i/(D'_i, C'_i)]$. $H_{i-1} \underset{s_i}{\to} H_i$ is called a *derivation step*, and a sequence of such derivation steps is called a *derivation*. □

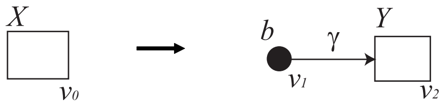

Fig. 2 An example of a production

An example of a production is shown in Figure 2. In the figure, a box is a nonterminal node and a filled circle is a terminal node. X, Y, and b are node labels and v_0, v_1, and v_2 are node IDs. Nodes with the same node label can appear in a graph, while nodes with same node ID will never appear in a graph. The production of Figure 2 indicates that after the removal of a nonterminal node with label X, embed the graph consisting 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 $(a, \alpha/\beta, v_1, in)$, but this connection instruction is not described in the notation of Figure 2.

In Figure 3, the production of Figure 2 and its connection instruction are drawn simultaneously. The large box in Figure 3 indicates the left-hand side, and two nodes with label b and Y are on the right side of the production of Figure 2. Node labels and edge labels that are described outside of the large box indicate connection instruction such that $(a, \alpha/\beta, v_1, in)$.

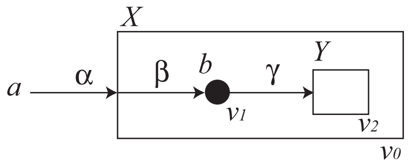

Fig. 3 An example of a production with the connection relation

An example of application of the production is shown in Figure 4. In Figure 4, $H = (V_H, E_H, \lambda_H)$ is a graph with $V_H = \{n_1, n_2\}$, $E_H = \{(n_1, \alpha, n_2)\}$, $\lambda_H(n_1) = a$, 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 $C' = \{(a, \alpha/\beta, n_3, in)\}$.

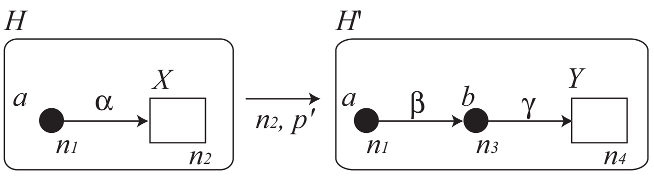

Fig. 4 An example of applying a production rule

We say that H is the *host graph* and H' is the *resulting graph*, X is the *mother node* in Figure 4, the graph consisting of terminal node with label b and the nonterminal node with label Y in Figure 2 is the *daughter 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 5. ([11], [12]) An *Attribute edNCE Graph Grammar* is a three-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 $p = X_0 \to (D, C) \in P$ 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. Behavioral diagrams are used to describe the behavior of what is being modeled and include use-case, activity, and state-machine diagrams.

Structural 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.

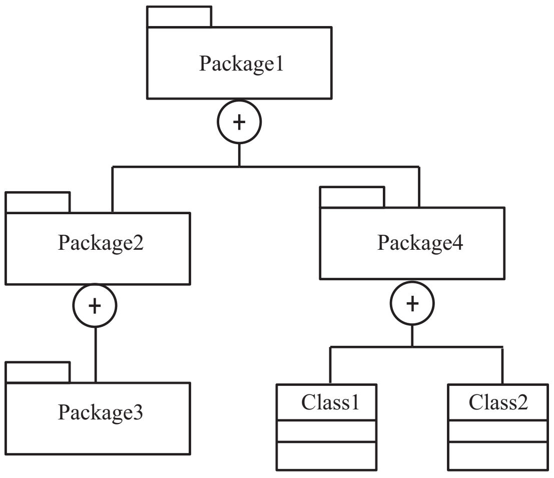

Fig. 5 An example of a package diagram

Figure 5 shows an example of a package diagram. The box with a rectangle at the upper left indicates a package. The box with three compartments is a class. Each of the three parts indicates its class name, its attribute, and its methods from top to the bottom. A plus with a circle is used to represent which components the package contains. Package 1 contains Package 2 and Package 4, and Package 4 contains Class 1 and Class 2.

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 6. The Graph Grammar for Package Diagrams (GGPD), for UML package diagrams, is a six-tuple $GGPD$ = (Σ_{PD}, Δ_{PD}, Γ_{PD}, Ω_{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, $\Gamma_{PD} = \{ * \}$, $\Omega_{PD} = \{ * \}$, $P_{PD} = \{ P_1, ..., P_{17} \}$ is a finite set of production rules, and $S_{PD} = \{ S \}$ is the initial non-terminal. □

The GGPD generates package hierarchy diagrams. Terminal nodes generated by GGPD have the following node labels: rop (root of package), sp (single package), lep (left side package), rip (right side package), mip (package located between lep and rip), lec (left side class), ric (right side class), and mic (class located between lec and ric).

GGPD is a context-free grammar and there are 17 production rules. An example of GGPD production rule is shown in Figure 6.

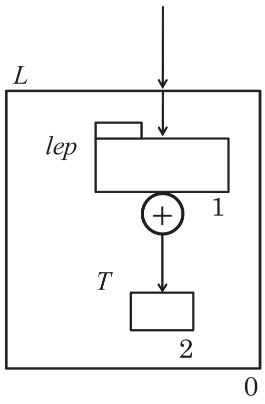

Fig. 6 An Example of a production rule of GGPD

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 a capitalized label indicates a nonterminal node, and a node with an uncapitalized label indicates a terminal node. Our grammar generates directed graphs. However, we drew the graphs without arrows as we assumed the direction of each edge was from the top down.

We omit descriptions of all of the production rules because of space limitations.

3.2 Example of Derivation

Figure 7 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, which 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 corresponds to UML package diagrams. In this case, we can obtain graph G_9.

We can obtain a derivation tree from a derivation sequence of production. Figure 8 shows the derivation tree corresponding to Figure 7. In Figure 8, the labels show the names of the production rules.

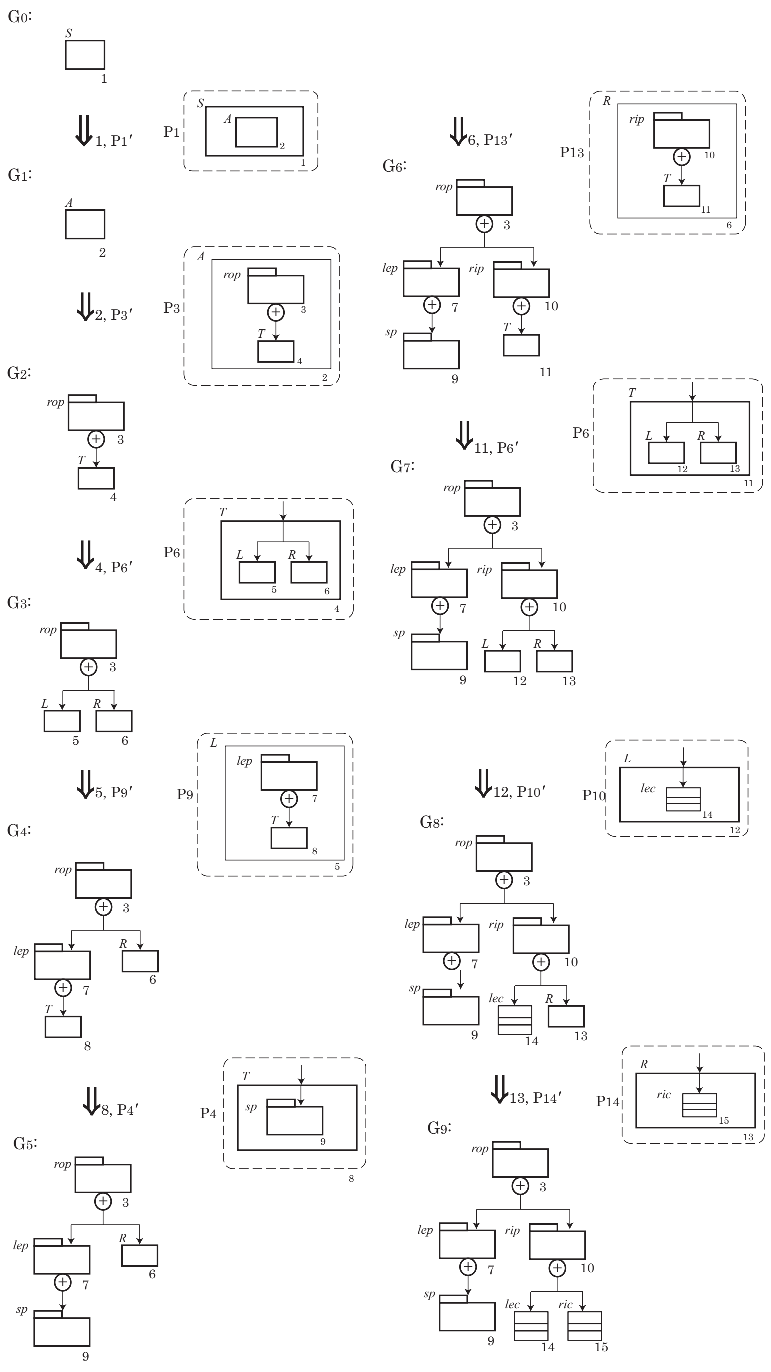
Fig. 7 An example of a GGPD derivation

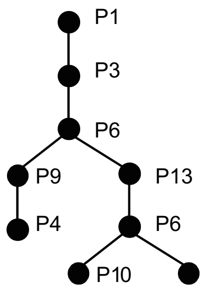

Fig. 8 A derivation tree corresponding to the tree in Figure 7

4. Document Generation

In this section, we explain some attributes and semantic rules for document generation. In this paper, we target a documentation that has diagrams and explanations that correspond to diagrams. We generate documents by attribute evaluation that can be executed automatically on derivation trees.

We can obtain derivation trees after generating diagrams based on our grammar. Figure 8 shows an example of a derivation tree. In the Figure, for example, Production 3 (P3) is applied to a node that was generated by Production 1 (P1). Derivation trees describe a process of applying productions.

Every node generated by productions of GGPD has some attributes. An attribute has two types of values called inherited attributes and synthesized attributes. Attribute values are obtained by calculating semantic rules on derivation trees. Values of inherited attributes are calculated by top-down on derivation trees, and bottom-up calculation generates values of synthesized attributes.

Figure 9 shows an example of production rules and their corresponding semantic rules. The upper part of Figure 9 indicates a production rule and the lower part shows semantic rules. This

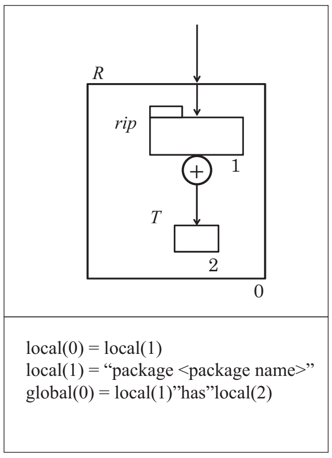
R
rip
1
T
2
0
local(0) = local(1)
local(1) = "package <package name>"
global(0) = local(1)"has"local(2)

Fig. 9 An Example of an attribute for document generation of GGPD

production rule rewrites a nonterminal node with node label R to a graph consisting of terminal node rip and nonterminal node T.

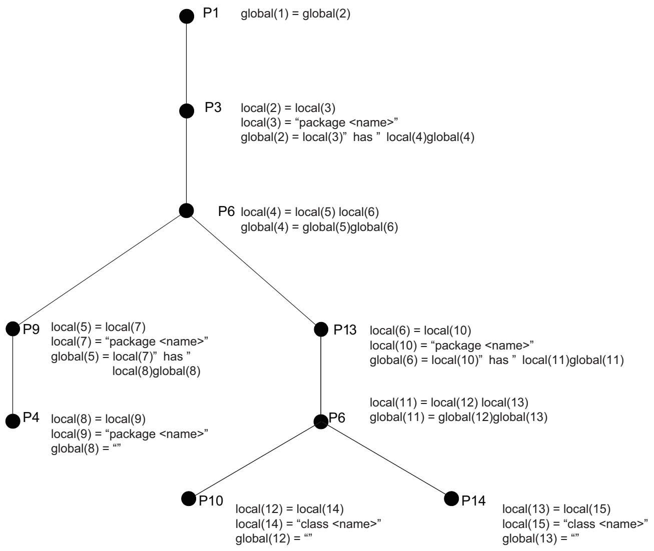
P1 global(1) = global(2)
P3 local(2) = local(3)
local(3) = "package <name>"
global(2) = local(3)" has " local(4)global(4)
P6 local(4) = local(5) local(6)
global(4) = global(5)global(6)
P9 local(5) = local(7)
local(7) = "package <name>"
global(5) = local(7)" has " local(8)global(8)
P13 local(6) = local(10)
local(10) = "package <name>"
global(6) = local(10)" has " local(11)global(11)
P4 local(8) = local(9)
local(9) = "package <name>"
global(8) = ""
P6 local(11) = local(12) local(13)
global(11) = global(12)global(13)
P10 local(12) = local(14)
local(14) = "class <name>"
global(12) = ""
P14 local(13) = local(15)
local(15) = "class <name>"
global(13) = ""

Fig. 10 A derivation tree with document generation attributes

In the semantic rules part, we then find three semantic rules that process the values of local and global attributes. The global attribute preserves the entire explanation of the diagrams. The local attribute retains local information such as parent and child relations. These attributes are categorized as synthesized attributes.

In Figure 9, local(1) stores node name of the node with ID 1 (in this case, package diagram's name); the value of local(1) is assigned to local(0).

Figure 10 shows an example of a derivation tree with document generation semantic rules corresponding to Figure 7.

In order to obtain documentation, attribute evaluation is processed following a bottom up order $P_4 \rightarrow P_9 \rightarrow P_{10} \rightarrow P_{14} \rightarrow P_6 \rightarrow P_{13} \rightarrow P_6 \rightarrow P_3 \rightarrow P_1$.

First, P_4 located at the lower left on the derivation tree is processed. In this case, we assume that the package name is "package 9," which is substituted for the local attribute of the node with ID 9. Local(9) is substituted for local(8). Null is substituted for global(9)

Next, P_9 is processed. Here, package name is substituted for local(7), and local(5) has the value of local(7). The global(5) stores a partial explanation of diagrams. In this case, global(5) holds information that node with ID 7 in Figure 7 has packages. The global(5) has the following sentence: "package 7 has package 9".

Global value is similarly obtained from the above procedure. The global(1) explains the entire diagram; in this example, global(1) is explained in Figure 11.

Since semantic rules are declarative defined, we can obtain an explanation corresponding to diagrams if once we define the semantic rules.

Package 2 has package 3.
Package 4 has class 5, class6.
Package 1 has package 2, package 4.

Fig. 11 An example of obtained explanation

5. Conclusion

In this paper, we have defined attributes and semantic rules for a graph grammar for UML package diagrams. We can generate documents corresponding to diagrams automatically according to a declarative definition.

A future issue for study is the synchronization between diagram and documentation and the implementation of systems that can generate documents with animation. We would also like to construct a graph grammar for other diagrams in UML.

References

[1] Takaaki Goto, Tetsuro Nishino, Kensei Tsuchida, "An Attribute Graph Grammar for UML Package Diagrams and its Applications," in *Proceedings of The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, Volume II*, 2011, pp. 693–698.

[2] L. Kotulski and D. Dymek, "On the Modeling Timing Behavior of the System with UML(VR)," in *Computational Science ICCS 2008*, ser. Lecture Notes in Computer Science, vol. 5101, 2008, pp. 386–395.

[3] F. Hermann, H. Ehrig, and G. Taentzer, "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, April 2008.

[4] 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.

[5] D. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications," in *Computer Performance Evaluation: Modelling Techniques and Tools*, ser. Lecture Notes in Computer Science, vol. 2324, 2002, pp. 183–204.

[6] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52.

[7] F. Meziane, N. Athanasakis, and S. Ananiadou, "Generating natural

language specifications from uml class diagrams," *Requirements Engineering*, vol. 13, pp. 1–18, 2008, 10.1007/s00766-007-0054-0.

[8] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," *SIGSOFT Softw. Eng. Notes*, vol. 27, pp. 229–239, July 2002.

[9] G. Rozenberg, *Handbook of Graph Grammar and Computing by Graph Transformation Volume 1*. World Scientific Publishing, 1997.

[10] M. Kaul, "Practical applications of precedence graph grammars," in *Graph Grammars and Their Application to Computer Science*, ser. LNCS 291, 1986, pp. 326–342.

[11] T. Nishino, "Attribute Graph Grammars with Applications to Hichart Program Chart Editors," in *Advances in Software Science and Technology*, vol. 1, 1989, pp. 89–104.

[12] T. Arita, K. Sugita, K. Tsuchida, and T. Yaku, "Syntactic Tabular Form Processing by Precedence Attribute Graph Grammars," in *Proc. IASTED Applied Informatics 2001*, 2001, pp. 637–642.