Attributed Context-Sensitive Graph Grammars

Luka Fürl
University of Ljubljana, Faculty of Computer and Information Science, Tržaška cesta 25, SI-1000 Ljubljana, Slovenia
E-mail: luka.fuerst@fri.uni-lj.si

Abstract. The paper introduces a concept of attributed context-sensitive graph grammars. The graph grammars are a graphical generalization of the textual grammars and can thus be used to specify the syntax of graphical programming or modeling languages. The attributed graph grammars extend the basic graph grammars with definitions of attributes and the associated attribute evaluation rules. By analogy to the attributed textual grammars, the purpose of the attributes and the rules is to define the semantic elements of a graphical language. The introduced concept is illustrated by an example of a grammar-driven conversion of flowcharts to the equivalent C code. The presented example might find its use in introductory programming courses.

Keywords: graph grammar, graphical language, visual language, semantics, attributed grammar

1 INTRODUCTION

The set of all syntactically valid programs in a textual programming language such as Pascal or C can be specified by a textual grammar, i.e., a formal system for generating textual strings based on a set of string replacement rules (productions). However, a grammar does not suffice for representing semantic elements such as variable scope or loop behavior. Such elements can be defined by attributed grammars [1], [2], a formalism that extends the ordinary grammars by a set of attributes assigned to individual grammar symbols and a set of production-specific rules for evaluating the attributes.

In this paper, we present how the attributes and attribute evaluation rules can be formulated within a graph grammar framework. The graph grammars are formal systems for generating graph sets and can thus be used for specifying the syntax of graphical languages [3]. We focus on the context-sensitive graph grammars, since they can represent a broader set of graphical languages than their context-free counterparts [4].

The attributed graph grammars remain an under-researched area. Their applications have been scarce; let us mention image parsing [5], music recognition [6] and graph drawing [7]. However, none of these approaches employs the concept of the attributed context-sensitive graph grammars in an analogous way to the attributed textual grammars. In our recent paper [8], we did use such a concept, but without a formal treatment. In this paper, we define the attributed context-sensitive graph grammars formally and provide a clarification example that might be applicable to introductory programming courses.

The rest of this paper is structured as follows. In Section 2, we define the context-sensitive graph grammars. In Section 3, we present our running example. Section 4 introduces the attributed context-sensitive graph grammars, and Section 5 concludes the paper.

2 CONTEXT-SENSITIVE GRAPH GRAMMARS

For a graph $G$, let $V[G]$, $E[G]$, and $X[G] = V[G] \cup E[G]$ denote the sets of its vertices, edges, and elements, respectively. Let $s(e)$ and $t(e)$ denote the source vertex and the target vertex, respectively, of an edge $e \in E[G]$. A graph $G$ is a subgraph of a graph $H$ (denoted $G \subseteq H$) if $V[G] \subseteq V[H]$ and $E[G] \subseteq E[H]$. A homomorphism $h: G \to H$ is a vertex-to-vertex and edge-to-edge map-
The graphs $L$ production

28 FÜRST

The notation $G$ graph

sensitive graph productions, and $(\ldots X \ldots)$ is called the $C$ possible common subgraph $C$ homomorphism.

is a bijective homomorphism whose inverse is also a homomorphism.

A context-sensitive graph production $p$ is a rule of the form $L ::= R$, where $L$ and $R$ are graphs with a possible common subgraph $C (C \subseteq L$ and $C \subseteq R)$. The graphs $L$ and $R$ are called the left-hand side (LHS) and the right-hand side (RHS), respectively, of the production $p$ (denoted $Lhs[p]$ and $Rhs[p]$). The graph $C$ is called the context of $p$ (denoted $Common[p]$). Let $X[p] = X[Lhs[p]] \cup X[Rhs[p]]$.

A context-sensitive graph grammar $GG$ is a quadruple $(\mathcal{N}, \mathcal{T}, \mathcal{P}, A)$, where $\mathcal{N}$ is a set of nonterminal labels, $\mathcal{T}$ is a set of terminal labels, $\mathcal{P}$ is a set of context-sensitive graph productions, and $A$ is a set of graphs called axioms. Each graph element in $\mathcal{P}$ and $A$ has to be labeled by a label from either $\mathcal{N}$ or $\mathcal{T}$.

To apply a context-sensitive graph production $p$ to a graph $G$, perform the following steps:

1. Find a monomorphism $h : Lhs[p] \rightarrow G$. Let $L' = h(Lhs[p]) \subseteq G$ and $C' = h(Common[p]) \subseteq L'$ denote the subgraphs of $G$ corresponding to $Lhs[p]$ and $Common[p]$, respectively.
2. Remove all elements $L' \setminus C'$ from $G$.
3. Add copies of the elements $Rhs[p] \setminus Common[p]$ to the resulting graph. The added elements have to be attached to the elements of $C'$ in the same way as the elements of $Rhs[p] \setminus Common[p]$ are attached to the elements of $Common[p]$.

The notation $G \xrightarrow{p} G'$ signifies that the graph $G'$ is the result of applying the production $p$ to the graph $G$.

A derivation of a graph $G$ in a context-sensitive graph grammar $GG$ is a sequence $A \xrightarrow{p_1} G_1 \xrightarrow{p_2} G_2 \ldots \xrightarrow{p_k} G_k = G$, where $A \in A[GG]$ and $p_1, \ldots, p_k \in \mathcal{P}[GG]$. A graph $G$ for which a derivation exists is derivable in $GG$.

The language of a grammar $GG$ (denoted $L(GG)$) is a set of all terminally labeled graphs derivable in $GG$. (A graph is terminally labeled if all its elements are labeled by labels from $\mathcal{T}[GG]$.) A parser is an algorithm that, for a graph $G$ and a grammar $GG$, determines whether $G \in L(GG)$ and produces a derivation of $G$ in $GG$ if this is the case. A parser for the context-sensitive graph grammars was introduced by Rekers and Schürr [4] and improved by Fuerst et al. [9].

### 3 Running Example

A grammar of flowcharts denoted $GG_{FC}$ will serve as a running example for the rest of this paper. For the grammar $GG_{FC}, \mathcal{N}[GG_{FC}] = \{ x \}$ and $\mathcal{T}[GG_{FC}] = \{ o, Cond, PrimStat, T, F, \}$. (For the context-sensitive grammar, the sets $\mathcal{B}$ and $\mathcal{R}$ are defined as follows:

- $\mathcal{B}$ is a set of triples $(p, x, A)$, where $p \in \mathcal{P}[GG], x \in \mathcal{X}[p]$, and $A$ is an attribute, a named variable assigned to a given graph element in a given production. An attribute $A$ for a graph element $x$ in a given production is denoted $x.A$. Let $B(x)$ denote the set of all attributes associated with the graph element $x$.)
### Table 1. Productions and attribute evaluation rules of the sample grammar for the flowcharts language.

<table>
<thead>
<tr>
<th>Production</th>
<th>Attribute evaluation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>( e_1 \cdot \text{code} = e_2 \cdot \text{code} \oplus e_3 \cdot \text{code} )</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>( e_1 \cdot \text{code} = \text{if isEmpty} (v_1 \cdot \text{label}) \text{ then } v_3 \cdot \text{code} \text{ else } v_1' \cdot \text{label} \oplus ' : ' \oplus v_3 \cdot \text{code} )</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>( e_1 \cdot \text{code} = \text{(if isEmpty} (v_1 \cdot \text{label}) \text{ then } \epsilon \text{ else } v_1' \cdot \text{label} \oplus ' : ') \oplus \text{' if (' } \oplus v_3 \cdot \text{code} \oplus \text{' } \text{'} \oplus e_2 \cdot \text{code} \oplus \text{' } \text{'} )</td>
</tr>
<tr>
<td>( p_4 )</td>
<td>( e_1 \cdot \text{code} = \text{(if isEmpty} (v_1 \cdot \text{label}) \text{ then } \epsilon \text{ else } v_1' \cdot \text{label} \oplus ' : ') \oplus \text{' if (' } \oplus v_3 \cdot \text{code} \oplus \text{' } \text{'} \oplus e_2 \cdot \text{code} \oplus \text{' } \text{'} )</td>
</tr>
<tr>
<td>( p_5 )</td>
<td>( e_1 \cdot \text{code} = \text{(if isEmpty} (v_1 \cdot \text{label}) \text{ then } \epsilon \text{ else } v_1' \cdot \text{label} \oplus ' : ') \oplus \text{' while (' } \oplus v_3 \cdot \text{code} \oplus \text{' } \text{'} \oplus e_2 \cdot \text{code} \oplus \text{' } \text{'} )</td>
</tr>
<tr>
<td>( p_6 )</td>
<td>( e_1 \cdot \text{code} = \text{(if isEmpty} (v_1 \cdot \text{label}) \text{ then } \epsilon \text{ else } v_1' \cdot \text{label} \oplus ' : ') \oplus \text{' do { (' } \oplus e_2 \cdot \text{code} \oplus \text{' } \text{'} \text{ while (' } \oplus v_3 \cdot \text{code} \oplus \text{' } \text{'} \text{') ; '} \text{) } )</td>
</tr>
</tbody>
</table>
| \( p_7 \)  | \( v_3 \cdot \text{label} = \text{if isEmpty} (v_3 \cdot \text{label}) \text{ then } \text{newLabel()} \text{ else } v_3 \cdot \text{label} \)  
\( e_1 \cdot \text{code} = \text{' goto ' } \oplus v_3' \cdot \text{label} \oplus \text{' ; } \) |
Figure 1. Sample derivation in the $GG_{FC}$ grammar.
• $R$ is a set of pairs $(p, R)$, where $p \in \mathcal{P}[GG]$ and $R$ is an attribute evaluation rule. An attribute evaluation rule for a production $p$ is a rule of the form $x_0.A_0 = \mathcal{f}(x_1.A_1, \ldots, x_k.A_k)$, where $x_0, x_1, \ldots, x_k \in \mathcal{X}[p]$ and $A_i \in \mathcal{B}(x_i)$ for all $i \in \{0, 1, \ldots, k\}$. The function $\mathcal{f}$ is called semantic function.

By analogy to the attributed textual grammars [2], the set of attributes for each production graph element can be partitioned into a set of synthesized attributes and a set of inherited attributes. The synthesized attributes are those associated with the graph elements on the production LHSs, while the inherited ones are those associated with the RHS elements. The set of the attribute evaluation rules must not contain any circular dependencies. If at least two attributes directly or indirectly depend on each other, their values cannot be computed.

The purpose of the attributes is to carry semantic information. In the usual case of compiling programs defined by the context-free grammars, the inherited attributes carry information about various symbols (variables, procedure declarations, etc.), while the synthesized attributes hold (intermediary) translations of the program or its parts. To obtain the ‘semantics’ of the program, the attributes have to be evaluated. In the case of the context-free grammars, the attributes are evaluated during the traversal of a derivation tree of the given program; the inherited attributes are evaluated on the way from the root to the leaves, while the synthesized ones are evaluated in the opposite direction. In the case of the context-sensitive graph grammars, derivations trees cannot be defined, however, since the production LHSs may contain more than one graph element. For this reason, the attributes can only be evaluated by traversing a derivation sequence of the input graph. The inherited attributes are evaluated during the left-to-right traversals of the sequence (from an axiom to the input graph), while the synthesized ones are evaluated during the right-to-left traversals. In general, several traversals may be required to evaluate all the attributes.

Let us formally describe a single attribute evaluation step. Let $p$ be a production, and let $\mathcal{X}[Lhs[p]] = \{a_1, \ldots, a_l\}$ and $\mathcal{X}[Rhs[p]] = \{b_1, \ldots, b_r\}$. Although the context elements occur on both sides of the production, we distinguish between their LHS and RHS occurrences. In a derivation step $G \xrightarrow{p} H$, let $L \subseteq G$ denote the subgraph corresponding to $Lhs[p]$, and let $R \subseteq H$ denote the subgraph corresponding to $Rhs[p]$. Let $\mathcal{X}[L] = \{x_1, \ldots, x_l\}$ and $\mathcal{X}[R] = \{y_1, \ldots, y_r\}$ such that $x_i (1 \leq i \leq l)$ corresponds to $a_i$, and $y_j (1 \leq j \leq r)$ corresponds to $b_j$. For each $j \in \{1, \ldots, r\}$, the following procedure is executed: if there is an inherited attribute $\mathcal{A} \in B(b_j)$ and a rule $b_j.\mathcal{A} = \mathcal{f}(a_{s_1}.A_{s_1}, \ldots, a_{s_t}.A_{s_t})$, and if all of $x_{s_1}, A_{s_1}, \ldots, x_{s_t}, A_{s_t}$ have already been evaluated, then evaluate $y_j.\mathcal{A}$ as $\mathcal{f}(x_{s_1}.A_{s_1}, \ldots, x_{s_t}, A_{s_t})$. The synthesized attributes are evaluated in the same fashion, only the LHS and RHS are reversed.

Let us proceed to our running example. The goal is to translate an arbitrary flowchart belonging to the language of the $GG_{FC}$ grammar to the equivalent C code. We define the following two attributes:

• code: This synthesized attribute is assigned to each $a$-labeled edge in the production set. It holds the C translation of the graphical statement derived from the associated edge $a$. The code attribute for the edge in the axiom graph holds the C translation of the entire flowchart.

• label: This inherited attribute is assigned to each empty vertex in the production set. At the beginning of the attribute evaluation procedure on a given flowchart, the value of the label attribute is initialized to an empty string for each empty vertex in the flowchart.

We do not translate the primitive statements and conditions, since we are interested only in the control flow. We merely assume that each PrimStat vertex and each Cond vertex has an associated code attribute that contains the C code for the primitive statement or condition represented by that vertex.

The attribute evaluation rules for individual productions are shown in the right column of Table 1. The rules employ the following special operators and functions:

- $\oplus$ denotes string concatenation ($a \oplus bc = abc$);
- $(if \ C \ then \ E_1 \ else \ E_2)$ returns $true$ iff the condition $C$ is satisfied, and $E_2$ if this is not the case;
- isEmpty($s$) returns true if the string $s$ is empty;
- newLabel() returns a unique label;
- $\epsilon$ represents an empty string.

Let us first consider the expressions involving the code attribute without paying attention to the label attributes. The C code for a sequence of statements (production $p_1$) is obtained by concatenating the translations of individual constituents. The code of a primitive statement is just the code associated with the vertex PrimStat. The code of a graphical if-then statement is obtained by embedding the code associated with the vertex Cond with and the edge $a$ on the production RHS into the if ... $\{\ldots\}$ framework. The code for the graphical if-then-else, while, and do-while statements is obtained in a similar fashion. The code for the graphical goto statement is goto $b$, where $b$ is a unique label assigned to the target vertex of the goto jump.

The code attributes are evaluated in the second (right-to-left) pass over the derivation of a given flowchart. During the first (left-to-right) pass, the inherited label attributes are evaluated. Every empty vertex that is the target of at least one goto statement receives a unique label. (The if-then expression within the v3.label := ... statement ensures that a label is reused if it already exists.) By the end of the first pass, all the goto targets have received the unique labels. When computing the
semantics is as-yet underexplored. Our future efforts might thus be directed towards strengthening the connection between the (meta)models, graph grammars and formalisms for specifying semantics.

## References


**Luka Fürst** received his B.Sc., M.Sc., and Ph.D. degrees from the Faculty of Computer and Information Science of the University of Ljubljana in 2004, 2007 and 2013, respectively. Since 2004, he has been employed with the same faculty, where he is currently working as a teaching assistant. His research interests include graph grammars, machine learning and graph algorithms in general.