ELEKTROTEHNIŠKI VESTNIK 81(1-2): 27–32, 2014
ORIGINAL SCIENTIFIC PAPER

Attributed Context-Sensitive Graph Grammars

Luka Fürst

*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

Kontekstno odvisne grafne gramatike s prilastki

Članek uvede koncept kontekstno odvisnih grafnih gramatik s prilastki. Grafne gramatike so grafna posplošitev tekstovnih gramatik, zato jih je mogoče uporabljati za podajanje sintakse grafnih programskih ali modelirnih jezikov. Grafne gramatike s prilastki razširjajo osnovne grafne gramatike z opredelitvijo prilastkov in pravil za računanje njihovih vrednosti. Po analogiji z besedilnimi gramatikami s prilastki je cilj prilastkov in pravil opredelitev semantičnih prvin grafnega jezika. Predstavljeni koncept ponazorimo s primerom grafnogramatične pretvorbe diagramov poteka v enakovredno programsko kodo v jeziku C. Prikazani primer je potencialno uporaben na uvodnih tečajih programiranja.

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 $\mathcal{V}[G]$, $\mathcal{E}[G]$, and $\mathcal{X}[G] = \mathcal{V}[G] \cup \mathcal{E}[G]$ 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 $e \in \mathcal{E}[G]$. Let $l(x)$ denote the label of an element $x \in \mathcal{X}[G]$. A graph G is a *subgraph* of a graph H (denoted $G \sqsubseteq H$) if $\mathcal{V}[G] \subseteq \mathcal{V}[H]$ and $\mathcal{E}[G] \subseteq \mathcal{E}[H]$. A *homomorphism* $h\colon G \to H$ is a vertex-to-vertex and edge-to-edge map-

Received 6 January 2014
Accepted 11 February 2014

ping that preserves labels and adjacencies: $l(h(x)) = l(x)$, $s(h(e)) = h(s(e))$, and $t(h(e)) = h(t(e))$ has to hold for each $x \in \mathcal{X}[G]$ and $e \in \mathcal{E}[G]$. A *monomorphism* is an injective homomorphism. An *isomorphism* 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 ($C \sqsubseteq L$ and $C \sqsubseteq R$). The graphs L and R are called the *left-hand side* (LHS) and the *right-hand side* (RHS), respectively, of the production p (denoted $\mathit{Lhs}[p]$ and $\mathit{Rhs}[p]$). The graph C is called the *context* of p (denoted $\mathit{Common}[p]$). Let $\mathcal{X}[p] = \mathcal{X}[\mathit{Lhs}[p]] \cup \mathcal{X}[\mathit{Rhs}[p]]$.

A *context-sensitive graph grammar* GG is a quadruple ($\mathcal{N}$, $\mathcal{T}$, $\mathcal{P}$, $\mathcal{A}$), 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 $\mathcal{A}$ is a set of graphs called *axioms*. Each graph element in $\mathcal{P}$ and $\mathcal{A}$ 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 $h\colon \mathit{Lhs}[p] \to G$. Let $L' = h(\mathit{Lhs}[p]) \sqsubseteq G$ and $C' = h(\mathit{Common}[p]) \sqsubseteq L'$ denote the subgraphs of G corresponding to $\mathit{Lhs}[p]$ and $\mathit{Common}[p]$, respectively.
2) Remove all elements $L' \setminus C'$ from G.
3) Add copies of the elements $\mathit{Rhs}[p] \setminus \mathit{Common}[p]$ to the resulting graph. The added elements have to be attached to the elements of C' in the same way as the elements of $\mathit{Rhs}[p] \setminus \mathit{Common}[p]$ are attached to the elements of $\mathit{Common}[p]$.

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 $A \in \mathcal{A}[GG]$ 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 $GG_{\mathrm{FC}}, \mathcal{N}[GG_{\mathrm{FC}}] = \{\mathsf{a}\}$ and $\mathcal{T}[GG_{\mathrm{FC}}] = \{\circ$, Cond, PrimStat, T, F, $|\}$, where $\circ$ and $|$ denote the virtual 'labels' of the unlabeled vertices and edges, respectively. The productions of GG_{FC} are displayed in the left column of Table 1; the right column will be explained later. For an easier reference, many graph elements are tagged by unique IDs (shown in square brackets, e.g., $[v_1]$ or $[e_2]$). The elements occurring on both sides of the individual productions, i.e., the elements of the Common sets, are grayed. These elements should actually have the same ID on both sides. However, since we will later have to distinguish between the LHS and RHS occurrences of the same element, we use primes (v_i') for the RHS occurrences. The grammar GG_{FC} contains a single axiom: this is the graph on the LHS of production p_1.

In the productions of the grammar GG_{FC}, the a-labeled edges represent graphical statements; in a derivation of a flowchart, every edge a develops into an elementary or composite graphical statement of a flowchart. The vertices $\circ$ (i.e., the unlabeled ones) represent the endpoints of individual statements. The PrimStat vertices represent primitive statements (non-control-flow ones), and the Cond vertices represent conditions. The edges T and F represent the 'true' and 'false' outcomes, respectively, of the associated conditions.

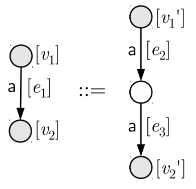
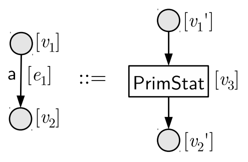
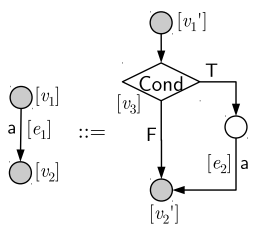
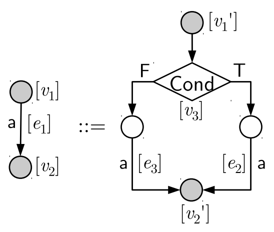
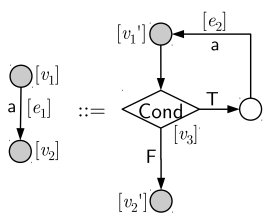
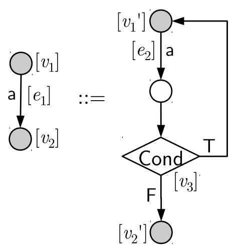
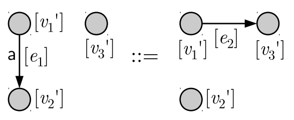
The axiom of the grammar GG_{FC} thus represents the fact that a flowchart comprises a single graphical statement. A graphical statement can take the form of a sequence of statements (production p_1), a primitive statement (p_2), an *if-then* statement (p_3), an *if-then-else* statement (p_4), a *while* statement (p_5), a *do-while* statement (p_6), or a *goto* statement (p_7). Since we are only interested in the control-flow statements, we grouped all the other types of the statements (assignments, reads, writes, etc.) under the 'primitive statements' category. Although the *goto* statement has been considered 'harmful' ever since the famous paper by Dijkstra [10], there are generally no objections against its use in flowcharts. In addition, as will be shown later, *goto* makes it possible to define a meaningful inherited attribute. A sample derivation in the grammar GG_{FC} is shown in Fig. 1.

4 Attributed Context-Sensitive Graph Grammars

We shall now introduce the central concept of this paper. An *attributed context-sensitive graph grammar* is a tuple (GG, $\mathcal{B}$, $\mathcal{R}$), where GG is a context-sensitive graph grammar, and 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 $\mathcal{B}(x)$ 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.*

	Production	Attribute evaluation rules
p_1		$e_1.\text{code} = e_2.\text{code} \oplus e_3.\text{code}$
p_2		$e_1.\text{code} = \textbf{if } \textit{isEmpty}(v_1.\text{label}) \textbf{ then } v_3.\text{code} \textbf{ else } v_1'.\text{label} \oplus \text{‘}\texttt{:}\text{’} \oplus v_3.\text{code}$
p_3		$e_1.\text{code} =$ $(\textbf{if } \textit{isEmpty}(v_1.\text{label}) \textbf{ then } \epsilon \textbf{ else } v_1'.\text{label} \oplus \text{‘}\texttt{:}\text{’})$ $\oplus$ $\text{‘}\texttt{if (}\text{’} \oplus v_3.\text{code} \oplus \text{‘}\texttt{) \{}\text{’} \oplus e_2.\text{code} \oplus \text{‘}\texttt{\}}\text{’}$
p_4		$e_1.\text{code} =$ $(\textbf{if } \textit{isEmpty}(v_1.\text{label}) \textbf{ then } \epsilon \textbf{ else } v_1'.\text{label} \oplus \text{‘}\texttt{:}\text{’})$ $\oplus$ $\text{‘}\texttt{if (}\text{’} \oplus v_3.\text{code} \oplus \text{‘}\texttt{) \{}\text{’} \oplus e_2.\text{code} \oplus \text{‘}\texttt{\}}\text{’}$ $\oplus\ \text{‘}\texttt{else \{}\text{’} \oplus e_3.\text{code} \oplus \text{‘}\texttt{\}}\text{’}$
p_5		$e_1.\text{code} =$ $(\textbf{if } \textit{isEmpty}(v_1.\text{label}) \textbf{ then } \epsilon \textbf{ else } v_1.'\text{label} \oplus \text{‘}\texttt{:}\text{’})$ $\oplus$ $\text{‘}\texttt{while (}\text{’} \oplus v_3.\text{code} \oplus \text{‘}\texttt{) \{}\text{’} \oplus e_2.\text{code} \oplus \text{‘}\texttt{\}}\text{’}$
p_6		$e_1.\text{code} =$ $(\textbf{if } \textit{isEmpty}(v_1.\text{label}) \textbf{ then } \epsilon \textbf{ else } v_1'.\text{label} \oplus \text{‘}\texttt{:}\text{’})$ $\oplus$ $\text{‘}\texttt{do \{}\text{’} \oplus e_2.\text{code} \oplus \text{‘}\texttt{\} while (}\text{’} \oplus v_3.\text{code} \oplus \text{‘}\texttt{);}\text{’}$
p_7		$v_3'.\text{label} = \textbf{if } \textit{isEmpty}(v_3.\text{label}) \textbf{ then } \textit{newLabel}() \textbf{ else } v_3.\text{label}$ $e_1.\text{code} = \text{‘}\texttt{goto }\text{’} \oplus v_3'.\text{label} \oplus \text{‘}\texttt{;}\text{’}$

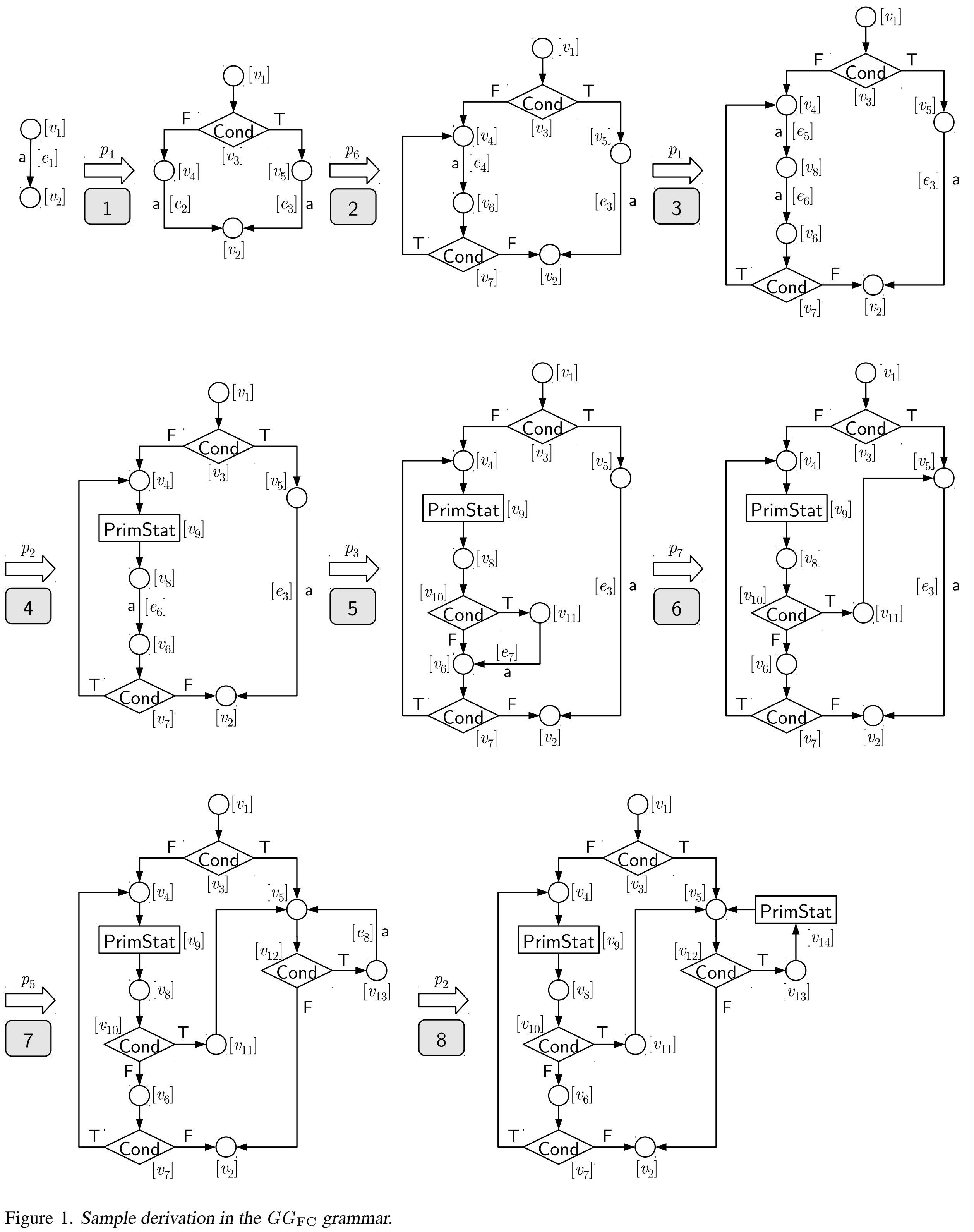


Figure 1. *Sample derivation in the GG_{FC} grammar.*

- $\mathcal{R}$ 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 $x_0.A_0 = f(x_1.A_1, \ldots, x_k.A_k)$, 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 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, derivation 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 $L \sqsubseteq G$ denote the subgraph corresponding to $Lhs[p]$, and let $R \sqsubseteq H$ 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 $A \in \mathcal{B}(b_j)$ and a rule $b_j.A = f(a_{s_1}.A_{t_1},\ \ldots,\ a_{s_l}.A_{t_l})$, and if all of $x_{s_1}.A_{t_1}, \ldots, x_{s_l}.A_{t_l}$ have already been evaluated, then evaluate $y_j.A$ as $f(x_{s_1}.A_{t_1},\ \ldots,\ x_{s_l}.A_{t_l})$. 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) equals E_1 if the condition C is satisfied, and E_2 if this is not the case;
- $\mathit{isEmpty}(s)$ returns `true` iff the string s is empty;
- $\mathit{newLabel}()$ returns a unique label;
- ϵ 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 and with 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 v_3.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

Table 2. *Attribute evaluation procedure for the derivation of Fig. 1.*

Derivation step	Evaluation
6	$v_5.\mathrm{label} = \texttt{a1}$
8	$e_8.\mathrm{code} = \overline{v_{14}}$
7	$e_3.\mathrm{code} = \texttt{a1: while (}\overline{v_{12}}\texttt{) \{ }\overline{v_{14}}\texttt{ \}}$
6	$e_7.\mathrm{code} = \texttt{goto a1;}$
5	$e_6.\mathrm{code} = \texttt{if (}\overline{v_{10}}\texttt{) \{ goto a1; \}}$
4	$e_5.\mathrm{code} = \overline{v_9}$
3	$e_4.\mathrm{code} = \overline{v_9}\texttt{ if (}\overline{v_{10}}\texttt{) \{ goto a1; \}}$
2	$e_2.\mathrm{code} = \texttt{do \{ }\overline{v_9}\texttt{ if (}\overline{v_{10}}\texttt{) \{ goto a1; \} \} while (}\overline{v_7}\texttt{);}$
1	$e_1.\mathrm{code} = \texttt{if (}\overline{v_3}\texttt{) \{ a1: while (}\overline{v_{12}}\texttt{) \{ }\overline{v_{14}}\texttt{ \} \}}$
	$\texttt{else \{ do \{ }\overline{v_9}\texttt{ if (}\overline{v_{10}}\texttt{) \{ goto a1; \} \} while (}\overline{v_7}\texttt{); \}}$

code attribute for a statement, we check whether the empty vertex marking the beginning of the statement has a nonempty label; if it has, it must be the target of at least one *goto*, and so we have to prepend the label and a colon to the code for the statement.

Table 2 shows the attribute evaluation procedure for the derivation of Fig. 1. To simplify the notation, we write $\overline{v_i}$ instead of $v_i.\mathrm{code}$. At the end, the code attribute for the edge in the axiom graph ($e_1.\mathrm{code}$) contains the translation of the entire flowchart.

5 Conclusion

We introduce a concept of the attributed context-sensitive graph grammars as a natural extension of the attributed textual grammars. If the syntax of a graph language is specified by a graph grammar, its semantics can be defined by the attributes attached to individual grammar elements and by the rules for evaluating the attributes during the traversal of a derivation of a given graph. As an example, we define a graph grammar of flowcharts and specify a set of the attributes and attribute evaluation rules used in translating individual flowcharts into the C code. The 'semantics' of a particular flowchart is thus defined as the C code that represents the same control flow as the flowchart. The presented example can be incorporated into a visual tool. Such a tool can be beneficial in introductory programming courses, where control structures are often presented by flowcharts. By translating the flowcharts to an equivalent textual code, the instructor can show the connection between a graphical and a textual representation of the control flow.

Recently, we proposed a novel method to convert metamodels (a declarative formalism for defining graph sets) to the equivalent context-sensitive graph grammars [8]. We showed that the semantics of models conforming to a given metamodel can be defined by an attributed graph grammar. However, the concept of the model 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

[1] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[2] J. Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[3] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg, and U. Montanari, editors. *Handbook of graph grammars and computing by graph transformation (Vols. 1.–3.).* World Scientific, 1997–1999.

[4] J. Rekers and A. Schürr. Defining and parsing visual languages with Layered Graph Grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[5] F. Han and S. Chun Zhu. Bottom-up/top-down image parsing with attribute grammar. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(1):59–73, 2009.

[6] S. Baumann. A simplified attributed graph grammar for high-level music recognition. In *Intl. Conf. on Document Analysis and Recognition, Montreal, Canada*, pages 1080–1083. IEEE, 1995.

[7] G. Zinßmeister and C. McCreary. Drawing graphs with attribute graph grammars. In *Graph Grammars and Their Applications to Computer Science, Williamsburg, VA, USA*, pages 443–453. Springer, 1994.

[8] L. Fürst, M. Mernik, and V. Mahnič. Converting metamodels to graph grammars: doing without advanced graph grammar features. *Software and Systems Modeling*. (to be published; DOI: 10.1007/s10270-013-0380-2).

[9] L. Fürst, M. Mernik, and V. Mahnič. Improving the graph grammar parser of Rekers and Schürr. *IET Software*, 5(2):246–261, 2011.

[10] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

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.