How to GRASP GOOD

Marc Andries
Leiden University
Dept. of Comp. Science
Niels Bohrweg 1
2333 CA Leiden
The Netherlands
andries@wi.leidenuniv.nl

Rudolf Freund
Technical University of Vienna
Inst. for Computer Languages
Resselgasse 3
A-1040 Vienna
Austria
freund@csdec1.tuwien.ac.at

Brigitte Haberstroh
Technical University of Vienna
Inst. for Information Systems
Paniglgasse 16
A-1040 Vienna
Austria
haber@vexpert.dbai.tuwien.ac.at

Abstract

The efforts of the past decade to provide database designers and users with more powerful tools for both modeling the considered application domain and manipulating this model, has resulted in a variety of declarative data languages of usually high complexity. In view of the implementation and optimization of such languages, it is often beneficial to use a second language as an intermediate, machine independent implementation platform. In this paper, we describe the mapping of the GOOD (**g**raph- and **o**bject-**o**riented **d**atabase) language to such an intermediate platform; as the GOOD language is based on graph rewriting, we use the novel graph grammar formalism of GRASPs (**a**ttributed **p**rogrammed **g**raph g**r**ammars with **s**et productions) for this purpose.

1 Introduction

The unability to adequately model in a database complex configurations of the real-world entities relevant to some application domain, is since long being considered as one of the major drawbacks of the relational database model. This observation caused the emergence of a variety of new paradigms and models, including nested relations, semantic and object-oriented database models. Soon it was recognized that the data representation part of these novel models allowed for a natural graphical representation in the spirit of the Entity Relationship model, by representing entities as nodes and relationships as edges [5, 12]. In

the late eighties, this in turn inspired a number of researchers to develop *graph-oriented* database models, in which notions from graph theory are used to uniformly define not only the data representation part of the model, but also their data *manipulation* languages [4, 7, 8, 13, 14, 18].

If data is modeled by means of graphs, manipulating data turns down to graph rewriting. Hence, graph grammars [16] offer themselves as a natural framework for the manipulation of graph-modeled data. The set of graph-oriented database operations to be presented in this paper, is derived from the **g**raph- and **o**bject-**o**riented **d**atabase model GOOD [13, 17]. In [3, 20], the graph grammar-like language of GOOD was shown to satisfy the well established completeness and consistency criterion for database manipulation languages called *BP-completeness* [6]. Briefly, a language that satisfies this criterion is capable of expressing exactly all database transformations that manipulate data items (i.e., nodes of the database graph) only on the basis of their relationships to other data items which are explicitly represented in the database (by means of the edges of the database graph).

The principle of BP-completeness is related to the typical set-oriented and associative nature of database query languages. This nature has its implications on the semantics that should be assigned to graph grammar productions in the context of databases. Usually, data retrieval and update operations take the form of "`apply` retrieval/update `to all` <data> `satisfying` <condition>" (cfr. the syntax of SQL-operations). In terms of the pattern matching paradigm underlying graph grammar productions, this becomes "`apply` retrieval/update `to all matchings of` <pattern> `in parallel`".

Attributing such a fully deterministic semantics to graph grammar productions unfortunately results in operations of high complexity, at least when compared to the "local" (and hence non-deterministic) semantics commonly used in graph grammar models. In view of the implementation of graphical database enduser interface tools, supporting language constructs providing manipulation facilities along the lines of graph-oriented database models like GOOD, this complexity calls for the introduction of an intermediate implementation level, e.g., for purposes of optimization. The fact that general purpose graph storage systems are beginning to emerge (see e.g., [15]), allows us to stay within the realm of graph

rewriting systems for our choice of this intermediate level. Hence, in this paper we use **a**ttributed **p**rogrammed **g**raph **gr**ammars with **s**et productions (GRASPs), an extended version of the model of attributed programmed graph grammars introduced in [9].

Attributed programmed graph grammars are a universal graph programming language. The programmed use of only a few types of locally operating graph productions allows efficient implementations of this graph grammar model. In [10] attributed programmed graph grammars have already been proved to be an adequate tool for user interface development. The correctness of the examples described there as well as of the examples presented in this paper was tested by using an interpreter for GRASPs written in SMALLTALK.

The rest of this paper is organized as follows. In Section 2 we introduce the concepts adapted from GOOD. In Section 3, we introduce the concepts of the new model of GRASPs. In Section 4, the simulations of the GOOD operations by equivalent GRASPs are outlined. In Section 5, we elaborate on the handling of node attributes (used to represent atomic values in the database), and briefly consider the matter of GRASP optimization. Finally, Section 6 contains conclusions and aspects of future work.

2 The Graph- and Object-Oriented Database Model GOOD

In this section, we present a simplified version of a subset of GOOD, concentrating only on those concepts required for the outline of the mapping to GRASPs.

A *database scheme* is typically used to specify a number of structural constraints on the actual contents of the database, which in turn is commonly referred to as the *database instance*. In GOOD, both scheme and instance are represented as *directed, labeled graphs*.

Definition 2.1 (Scheme) A *scheme* is a quadruple $\mathcal{S} = (V, W, A, F)$ with V a finite set of node labels, W a finite set of edge labels, A a set of attribute values, and $F \subseteq V \times W \times V$. If $F = V \times W \times V$, we write $\mathcal{S} = (V, W, A)$. If, moreover, $A = \emptyset$, we write $\mathcal{S} = (V, W)$.

The elements of V may be considered to represent class names, while the elements of W represent names for properties and relationships. F then specifies what relationships are allowed to exist between members of certain classes.

Definition 2.2 ((Attributed) Graph) Let V and W be sets of labels and A a set of values (including the value nil). A *graph* over (V, W) is a triple $g = (N, n, E)$ where N is a finite set of nodes, $n : N \to V$ is the node-labeling function, and $E \subseteq N \times W \times N$ is a finite set of directed, labeled edges. The set of all graphs over (V, W) is denoted by $\gamma(V, W)$. An *attributed graph* over (V, W, A) is a quadruple $g = (N, n, a, E)$, where $g^* = (N, n, E) \in \gamma(V, W)$ is the underlying graph and $a : N \to A$ is the attribution function. The set of all attributed graphs over (V, W, A) is denoted $\gamma(V, W, A)$.

Example 2.1 Let $\mathcal{S} = (V, W, A, F)$ be a scheme. Then F can be naturally represented as the graph (V, id_V, F) over (V, W), where id_V is the identity function on V.

Definition 2.3 (Instance) Let $\mathcal{S} = (V, W, A, F)$ be a scheme. An *instance* over $\mathcal{S}$ is an attributed graph $\mathcal{I} = (N, n, a, E) \in \gamma(V, W, A)$, such that $(x, \alpha, y) \in E$ implies $(n(x), \alpha, n(y)) \in F$.

Note that the requirement that $(n(x), \alpha, n(y))$ be an element of F ensures that an instance satisfies the structural constraints of the scheme.

We now define syntax and semantics of the GOOD-operations to be used in this paper.

Definition 2.4 (Pattern) A *pattern* over a scheme $\mathcal{S}$ is an instance over $\mathcal{S}$ that may contain nodes without specified attribute value.

Definition 2.5 (Matching) Let $\mathcal{S}$ be a scheme, let $\mathcal{I}$ be an instance over $\mathcal{S}$ and let $\mathcal{J}$ be a pattern over $\mathcal{S}$. A *matching* of $\mathcal{J}$ in $\mathcal{I}$ is an injective mapping from the nodes in $\mathcal{J}$ to the nodes in $\mathcal{I}$ preserving labels and edges, as well as the attributes specified in $\mathcal{J}$.

Example 2.2 In order to elucidate the concepts presented in this paper, we shall use a simple example database containing information about operas, their title, their composer

and the year of their world premiere. The graph representation of a possible scheme $\mathcal{S}$ for such a database is presented in Figure 1, while a (very small) instance over this scheme is shown in Figure 2.

The pattern of Figure 3 represents the operas composed by Mozart. The reader can easily verify that this pattern has two matchings in the instance of Figure 2.

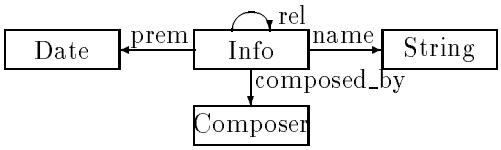

Figure 1: Opera Database Scheme

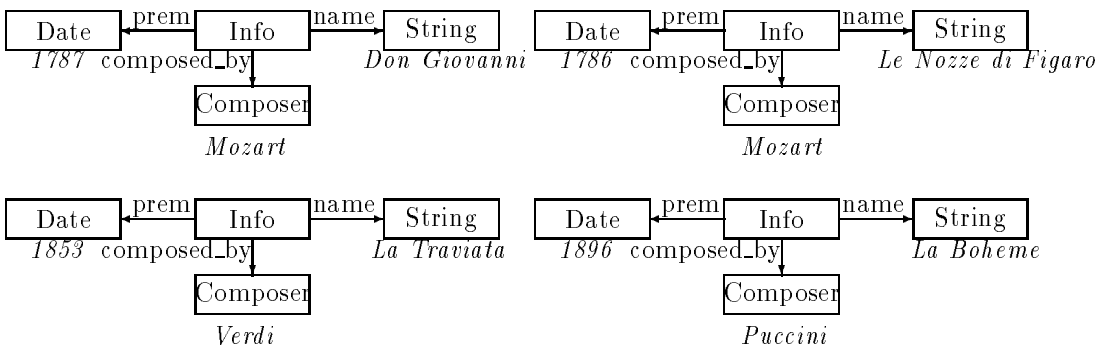

Figure 2: Opera Database Instance

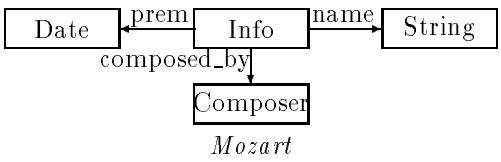

Figure 3: Pattern over the Opera Database

Definition 2.6 (GOOD operation) Let $\mathcal{S} = (V, W, A, F)$ be a scheme, let $\mathcal{I} = (N, n, a, E)$ be an instance over $\mathcal{S}$ and let $\mathcal{J} = (N', n', a', E')$ be a pattern over $\mathcal{S}$. Let $\mathcal{M}$ be the set of all matchings of $\mathcal{J}$ in $\mathcal{I}$. Four types of GOOD operations can be specified on a pattern. For each type, we now state the required parameters, as well as an algorithm that should be applied to $\mathcal{I}$ to obtain the result of the operation

Node Addition: Parameters: $x_1, \ldots, x_k \in N'$; $K \in V$; $a \in A$; $\alpha_1, \ldots, \alpha_m \in W$.

Algorithm:

for each *matching* $e \in \mathcal{M}$ **do**

if not exists *a node* y *labeled* K *attributed* a *in* $\mathcal{I}$ *with edges* $(y, \alpha_\ell, e(x_\ell))$, $(1 \leq \ell \leq k)$

then *add a new node* y' *labeled* K *attributed* a *to* $\mathcal{I}$ *with edges* $(y', \alpha_\ell, e(x_\ell))$, $(1 \leq \ell \leq k)$.

Edge Addition: Parameters: $x, x' \in N'$; $\alpha \in W$.

Algorithm:

for each *matching* $e \in \mathcal{M}$ **do**

$E(\mathcal{I}) := E(\mathcal{I}) \cup \{(e(x), \alpha, e(x'))\}$.

Node Deletion: Parameters: $x \in N$.

Algorithm:

for each *matching* $e \in \mathcal{M}$ **do**

$E(\mathcal{I}) := E(\mathcal{I}) - \{(e(x), \alpha, e(y)), (e(y), \alpha, e(x)) \mid \alpha \in W, y \in N\}$;

$N(\mathcal{I}) := N(\mathcal{I}) - \{e(x)\}$.

Edge Deletion: Parameters: $(x, \alpha, x') \in E'$.

Algorithm:

for each *matching* $e \in \mathcal{M}$ **do**

$E(\mathcal{I}) := E(\mathcal{I}) - \{(e(x), \alpha, e(x'))\}$.

By means of the examples presented in Section 5, the reader is invited to verify that the semantics of these four operations is well-defined, and deterministic up to the choice of new nodes in the resulting instance of a node addition.

3 GRASPs – Attributed Programmed Graph Grammars with Set Productions

The model of attributed elementary programmed graph grammars was introduced in [9]. It is based on the programmed use of elementary local graph productions, i.e., the insertion or deletion of nodes or edges, and the renaming of nodes and edges, chosen nondeterministically on the basis of as little context as possible. In the same article, the model was shown a universal graph programming language in the sense that the model is sufficiently powerful to generate any recursively enumerable language of attributed graphs

In this section, we introduce the new model of attributed programmed graph grammars with set productions (GRASPs for short), which is based on the programmed use of *sets* of these elementary graph productions.

Definition 3.1 (Set Production) Let λ be a special (*blank*) symbol, and let V, W be alphabets. Then we define $V_\lambda := V \cup \{\lambda\}$ and $W_\lambda := W \cup \{\lambda\}$. The set of node productions over (V, W) is defined as $R_N(V, W) := V_\lambda^2 - \{\lambda\}^2$. The set of edge productions over (V, W) is defined as $R_E(V, W) := (V \times W_\lambda \times V)^2 - (V \times \{\lambda\} \times V)^2$.

An element of $R(V, W) := R_N(V, W) \cup R_E(V, W)$ is called a *graph production*, while any subset of $R(V, W)$ is called a set of graph productions (or *set production* for short).

A set production $p \in R(V, W)$ is said to be applicable to a graph $g \in \gamma(V, W)$, if there exists a graph production $q \in p$ that is applicable to the graph g. The applicability of such a production q and the result of applying q to g — yielding a new graph $g' \in \gamma(V, W)$ — depends on the form of q:

1. $\underline{(\lambda, L)}$: add a new node with label L;

2. $\underline{(K, L)}$: change the label of a node from K to L;

3. $\underline{(K, \lambda)}$: delete an isolated node (i.e., without edges ending at it or leaving it) with label K;

4. $\underline{(K, \lambda, L; C, k, D)}$: add an edge with label k between a node x with label K and a node y with label L and change these node labels to C and D respectively, but only if (x, k, y) is not already an edge in g;

5. $\underline{(K, j, L; C, k, D)}$: change the label of an edge with label j between a node x with label K and a node y with label L to k and change these node labels to C and D respectively, but only if (x, k, y) is not already an edge in g;

6. $\underline{(K, j, L; C, \lambda, D)}$: delete an edge with label j between a node with label K and a node with label L and change these node labels to C and D respectively.

Observe that in the case of the edge productions of the form $(K, j, L; C, k, D)$, we may choose the same node x and y if $K = L$, but only if $C = D$ too.

Definition 3.2 (Attributed Programmed Graph Grammar with Set Productions) An *attributed programmed graph grammar with set productions* (GRASP) is a 6-tuple $G = (V, W, A, P, r_0, R_f)$ where

- V and W are alphabets for labeling nodes and edges, respectively, and A is a set of attributes;

- P is a finite set of rules $(r : p, \sigma(r), \varphi(r))$, where r is a label for the set production $p \subseteq R(V, W)$, $\varphi(r) \subseteq Lab(P)$ and $\sigma(r) \subseteq Lab(P) \times \mathcal{F}^{|p|}$, $\mathcal{F}$ is a finite set of (partial recursive) functions with arguments in A, and $Lab(P) = \{r \mid (r : p, \sigma(r), \varphi(r)) \in P\}$; if $p = \{q_i \mid 1 \leq i \leq m\}$, then the function f_i in an element $(r, f_1, \ldots, f_m)$ of $\sigma(r)$ is said to be *assigned* to the production q_i;

- $r_0 \in Lab(P)$ is the initial label (of the initial rule);

- $R_f \subseteq Lab(P)$ is a set of final labels (of final rules).

We define $\wp(r) := p$ for each rule in P.

The kind of function that can be assigned to a production depends on the type of production:

- For any $q_i \in \wp(r)$ of the form (λ, L), we allow only constant functions to be assigned;
- For any $q_i \in \wp(r)$ of the form (K, L) with $K, L \in V$, we assign functions $f_i : A \to A$ for every $(r', f_1, \ldots, f_m) \in \sigma(r)$;
- For any $q_i \in \wp(r)$ of the form (K, λ), no functions need to be assigned, hence we assume $f_i = \emptyset$ for every $(r', f_1, \ldots, f_m) \in \sigma(r)$;
- For any $q_i \in \wp(r)$ which is in $R_E(V, W)$, we assign functions $f_i = (f_{i,1}, f_{i,2})$ with $f_{i,1}, f_{i,2} : A^2 \to A$. Function $f_{i,1}$ (resp. $f_{i,2}$) computes the new attribute of the target (resp. source) of the edge affected by q_i, using the old attributes of both source and target of this edge.

For each $g, g' \in \gamma(V, W, A)$, the pair (g', r') is said to be *directly derivable* from (g, r) in G — abbreviated $(g, r) \vdash_G (g', r')$ — if either

1. $g' = g$, none of the productions in $\wp(r)$ can be applied to the graph g, and $r' \in \varphi(r)$ (we denote this as $(g, r) \vdash_G^N (g', r')$), or
2. g'^* is obtained from g^* by applying a graph production $q_i \in \wp(r), (r', f_1, \ldots, f_m) \in \sigma(r)$, and the attributes of the nodes affected by the application of the graph production q_i to g can be changed by the function f_i (i.e., they are in the domain of f_i), thus yielding g' (we denote this as $(g, r) \vdash_G^{(Y, f_i)} (g', r')$).

If for every graph production in $\wp(r)$ which is applicable to g^*, the assigned attribution function is not defined for the attributes under consideration, no further derivation from the pair (g, r) is possible.

An attributed graph g' is said to be *derivable* from an attributed graph g in G, if there exists a sequence of subsequent direct derivations, starting with the pair (g, r_0), and ending in (g', r_f) with $r_f \in R_f$.

A GRASP can easily be described by a control diagram, in which each rule $(r : p, \sigma(r), \varphi(r))$ is represented by a node labeled $r : p$ and edges leaving this node; these are either labeled by Y(es) and the corresponding attribution functions (unless these functions do not affect the attributes, in which case we omit them), or N(o), and end in the nodes representing the rules in $\sigma(r)$ and $\varphi(r)$ respectively. In addition, we use the special labels `begin` resp. `end` to replace the initial resp. the final rule.

Example 3.1 Let f be the identity function defined on the singleton $\{c\}$, where c is some given attribute value. Let g be the identity function defined on $A - \{c\}$. Then we can detect every node having c as attribute value, using the GRASP corresponding to the control diagram of Figure 4. These nodes are marked with an asterisk in rule m_1, while the same rule marks all other nodes with a quote. These quotes are subsequently removed in rule m_2.

Figure 4: GRASP detecting a given attribute

4 Mapping GOOD to GRASP

In this section, we describe the simulation of GOOD operations by graph grammars in the model of GRASPs, by mapping each of the GOOD operations to a GRASP.

The first problem is to determine all matchings of a given pattern $\mathcal{J}$ with nodes $\{u_1, \ldots, u_n\}$ in the following way: for each matching e, we add a new node labeled S with

outgoing edges labeled 1 through n, leading to the nodes $e(u_1)$ through $e(u_n)$ respectively. The simulation of this operation by means of a GRASP is tedious but straightforward, and hence ommitted. The main idea is to pick a "candidate" matching from the instance for the nodes $u_1 \ldots u_n$ and then to check the presence of the required edges between the chosen nodes. If this check fails, we backtrack and undo the last choice. Eventually, a viable candidate matching for each node is found and marked with an S-node as described above. The process is then iterated until no more matchings can be found. In the sequel, we shall denote the GRASP that looks for all matchings of a given pattern $\mathcal{J}$ by $G_{\mathcal{J}}$. For illustrations, we refer to Example 5.1 in Section 5.

We now outline how each of the four GOOD-operations can be translated into a corresponding GRASP.

Algorithm 4.1 (Node Addition) Let $\mathcal{J}$ be a pattern with nodes $\{u_1, \ldots, u_n\}$ with labels $n(u_i) = l_i$ $(1 \leq i \leq n)$ (for simplicity, we assume $i \neq j \Rightarrow l_i \neq l_j$). Let $u_1, \ldots, u_m$ $(m \leq n)$ be the nodes in the images of which the edges labeled $\alpha_1, \ldots, \alpha_m$ should arrive. For a particular matching e, the subgraph isomorphic to $\mathcal{J}$ is supposed to be marked previously by a node s_e labeled S with outgoing edges $(s_e, k, e(u_k)), 1 \leq k \leq n$.

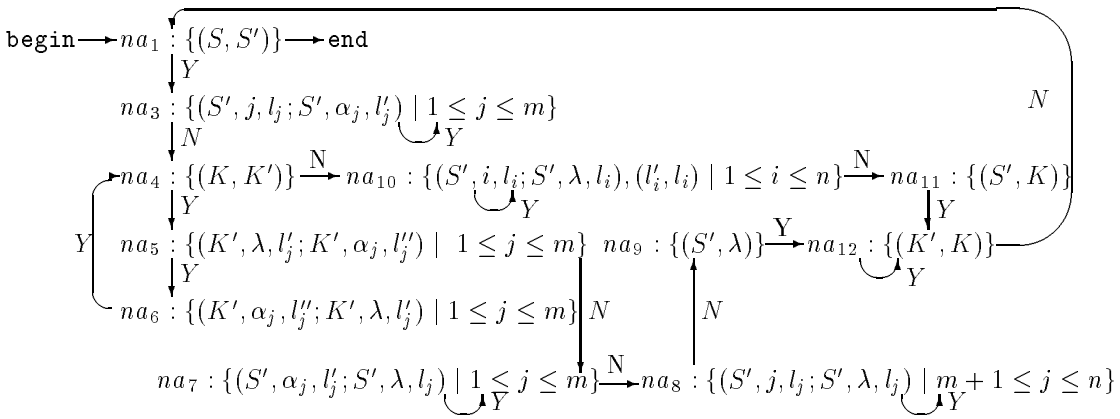

Figure 5: GRASP simulating a node addition

Using rule na_3 of the GRASP of Figure 5, we introduce new edges α_j by relabeling the corresponding edges leading from the node s_e to the node $e(u_j)$ and then, by rule na_5 we

check whether a new node with label K has to be inserted or not. If no new node has to be inserted because the subgraph we would insert is already part of the instance (e.g., because the desired node addition has already been performed by a previous matching), we use rules na_7 through na_9 to delete the edges leaving the node s_e and this node itself. Otherwise the addition is completed by rules na_{10} through na_{12}.

Algorithm 4.2 (Node Deletion) For the GRASP for node deletion depicted in Figure 6, we assume the nodes to be deleted have been marked with an edge labeled 1, leaving the nodes s_e representing the matchings e. By rule nd_1 these edges are deleted and the selected nodes are marked with a prime. Then all edges leaving resp. ending in such a selected node are deleted by rule nd_2, and by rule nd_3 the selected nodes are deleted themselves. Finally, rules nd_4 and nd_5 eliminate the remaining edges leaving the nodes s_e as well as these nodes themselves.

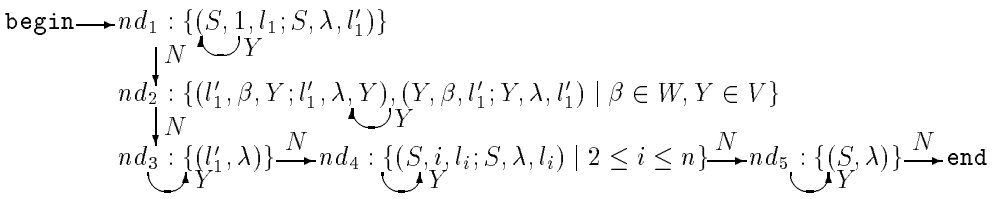

Figure 6: GRASP simulating a node deletion

Algorithm 4.3 (Edge Addition/Deletion) The GRASPs for edge addition and deletion are quite similar; they only differ in the single rule which executes the desired edge operation. We assume the edge to be added resp. deleted in each matching e leads from the node marked by an edge labeled 1 (leaving the corresponding node s_e marked S) to the node which is marked by an edge labeled 2 (also leaving s_e). In the GRASP of Figure 7 (showing the GRASP for edge addition) the rules 4 and 5 delete the edges labeled 1 and 2 and mark the corresponding nodes, which allows us to apply the desired edge operation in rule 6 (the GRASP for edge deletion is obtained by interchanging in this rule the role of α and λ). When this rule cannot be applied, it means that the desired edge operation

has already been caused by another matching, or that the edge already existed (in case of an addition). After having handled all matchings in this way, we delete the remaining edges labeled $i, 3 \leq i \leq n$ by using rule 2, and finally by rule 3 we delete the nodes s_e. Observe that in order to deal with loops, the GRASP has to be altered slightly. The obvious changes to the diagram are left as an exercise to the reader.

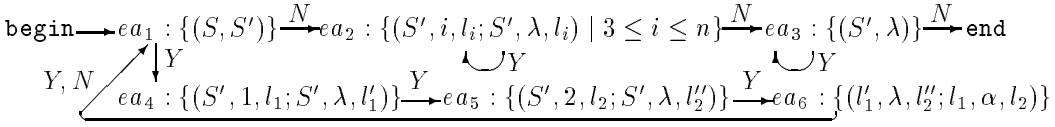

Figure 7: GRASP simulating an edge addition

5 Applications

In this section, we illustrate by means of examples on the Opera database introduced in Section 2, the handling of attributes by GRASPs, and show how matchings of arbitrary patterns may be marked, as described in the previous section. We also consider the matter of optimizing GRASP simulations.

For GOOD operations, we use the following simple graphical conventions: patterns are indicated in plain line, what is added is indicated with bold lines, and what is deleted is indicated with double lines.

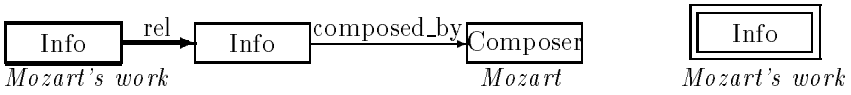

Figure 8: A Node Addition (left) and Deletion (right)

Example 5.1 Consider the node addition of Figure 8, left. The pattern of this operation selects all Info-nodes, linked by a composed_by-edge to a Composer-node with attribute

Mozart. For each such node, a new Info-node is added with attribute *Mozart's work*, linked to the existing Info-node with a rel-edge. Figure 9 shows the GRASP that marks all matchings of the given pattern. The following functions are used: f is the identity function defined on the singleton $\{Mozart\}$, g is the identity function defined on $A - \{Mozart\}$, h is the constant function mapping all attributes of A to *Mozart's work*, and Φ is the function with empty domain.

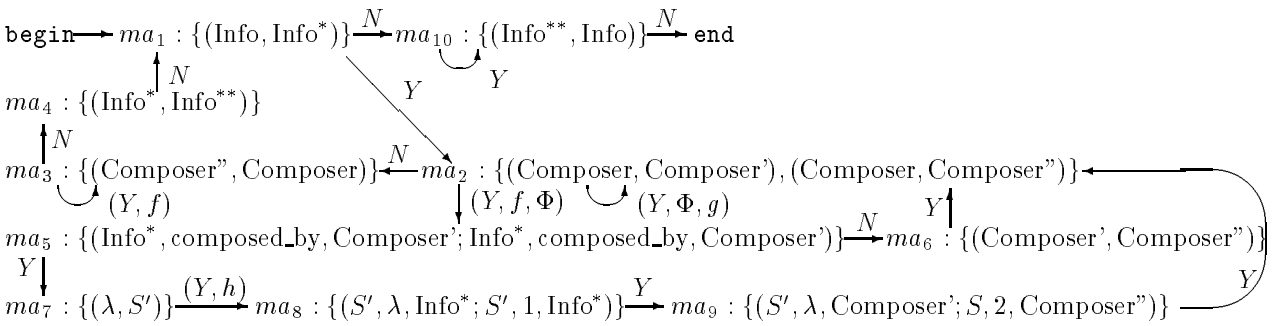

Figure 9: GRASP marking all matchings of a pattern

We first mark an Info-node with one asterisk by rule ma_1. Then by rule ma_2 we search for a Composer-node and mark it with one prime if it is attributed *Mozart*. The other Composer-nodes are marked with two primes. If the attribute is indeed *Mozart*, the pair of functions (f, Φ) takes care that control shifts to ma_5, otherwise we stay at rule ma_2. If moreover, a composed_by- edge leads from the selected Info*-node to the selected Composer-node, the rules ma_7 through ma_9 add a new S-node with the attribute *Mozart's work*, and with outgoing edges labeled 1 and 2 to respectively the chosen Info- and Composer-node. Combining this GRASP with an appropriate version of the GRASP for node addition of Figure 5, we finally get the GRASP for the GOOD-operation of Figure 8, left.

We can optimize the previous solution considerably in a similar way as selections can be pushed through joins in relational query optimization. Observing that for the node addition of Figure 8, left, we only have to select all the Info-nodes connected by a composed_by-edge to a Composer-node with attribute *Mozart*, we get the GRASP of Figure 10. The

attribution functions in this GRASP are f_2, which is the identity defined on $A \times \{Mozart\}$, g_2, which is the identity defined on $A \times (A - \{Mozart\})$, f_3, which is the identity defined on $\{Mozart's\ work\} \times A$, g_3, which is the identity defined on $(A - \{Mozart's\ work\}) \times A$, and h, the constant function mapping all attributes of A to *Mozart's work*. The "correct" Info-nodes are marked with an asterisk by rule a_1 (the other Info-nodes are marked with a prime and are relabeled by rule a_2). Rule a_3 relabels those Info*-nodes already related to an Info-node attributed *Mozart's work* by a preceding node addition. Finally, by rules a_4 through a_6, for each Info*-node n a new S-node m is added with attribute *Mozart's work*, as well as an edge (n, rel, m).

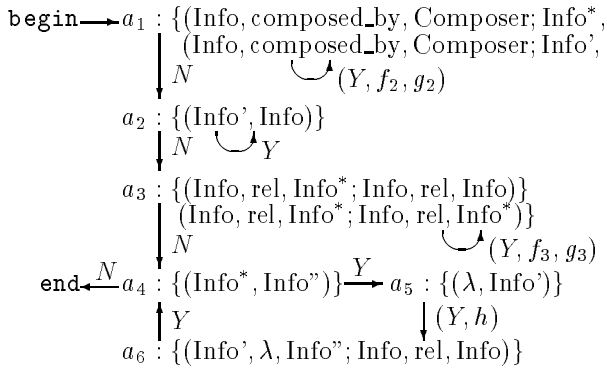

Figure 10: Optimized GRASP

In support of our claim that reorganizing a GRASP on the basis of structural information about the patterns of the simulated GOOD operation (as we did when transforming the combination of the GRASPs of Figures 5 and 9 into the single GRASP of Figure 10) may imply a relevant performance optimization, we now briefly make some considerations on the time-complexity of GRASPs.

Let n be the number of nodes in the instance graph. Then it can easily be verified that (using a standard representation for directed labeled graphs) productions in $R_N(V, W)$ (affecting a single node) have complexity $O(n)$, productions in $R_E(V, W)$ (affecting a single edge and the adjacent nodes) have complexity $O(n^2)$.

Applying these observations, it may be verified that the GRASP in Figure 9 has complexity $O(n^4)$ and therefore also the combination of the GRASPs of the Figures 5 and 9 has complexity $O(n^4)$, while the single (equivalent) GRASP of Figure 10 has complexity $O(n^2)$ only.

In essence, this considerable decrease in complexity is caused by the fact that we are dealing here with an operation on a simple pattern $\mathcal{J}$ (only one edge and its adjacent nodes), which allows us to search for an edge and its adjacent nodes only instead of choosing all possible combinations of two nodes and testing all the edges between these two nodes as has to be done in the general algorithm $G_{\mathcal{J}}$ for the pattern $\mathcal{J}$. In a similar way, for special patterns occurring in GOOD operations the corresponding GRASPs can be optimized in order to increase efficiency. E.g., for patterns consisting of a chain of edges only, similar improvements as described above may be obtained.

Other, more heuristic rules may be used in the construction of more efficient GRASPs for the execution of GOOD-operations:

- Based on the number of objects per class in the database under consideration (which is typical statistical information for OODBs), the node set of the pattern of the GOOD operation may be ordered such that objects of a class with few objects are looked for first. Otherwise, these "rare" nodes would have to be looked for over and over again, once for each node of a class with numerous objects;

- Another relevant statistical parameter is the expected number of relationships in which objects from a given class will participate (i.e., the number of edges leaving from or ending in nodes with a given label). The node set of the pattern should then be ordered such that nodes for which this number is small, are looked for first, which will minimize the total number of edges to be checked for;

- Similar statistical information may be used in conjunction with the above principle of optimization to scan for an edge and its adjacent nodes. Applying this principle implies the need to look for an ordering of the edges (instead of the nodes, as in the first item of this list), which may well be done on the basis of some statistics.

In conclusion, the translation of the high level GOOD operations into GRASPs allows the analysis of the complexity of database operations on a much more elementary formal level, yet without relying on a specific physical implementation.

6 Conclusions and Further Research

In this paper, we have shown that the strictly locally operating graph rewriting model of attributed programmed graph grammars with set productions (GRASPs) is capable of serving as an implementation platform for high level, set-oriented and deterministic database operations like those of the GOOD (graph- and object-oriented database) language. Moreover, we have explicated how, argumenting on a formal basis, the performance of GOOD operations can be optimized by investigating the control flow in the corresponding GRASP diagrams. Taking into account the results of this paper, the precompiler for GRASPs we have started to implement will also allow us to build up a prototype for a database managing system based on GOOD operations. This work should be seen in view of ongoing research at the University of Antwerp (UIA). A mapping of GOOD onto the relational model and algebra is serving as the basis for a prototype implementation of an actual user interface, based on GOOD [2, 11]. In our view, as mentioned in the Introduction, the graph-oriented nature of GOOD makes the choice of a graph-grammar model like GRASP as an implementation platform more natural.

References

[1] *Proceedings of the Ninth ACM Symposium on Principles of Database Systems.* ACM Press, 1990.

[2] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. Concepts for graph-oriented object manipulation. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *3rd International Conference on Extending Database Technology, Proceedings*, number 580 in Lecture Notes in Computer Science, pages 21–38, Berlin, 1992. Springer-Verlag.

[3] M. Andries and J. Paredaens. A language for generic graph-transformations. In Schmidt and Berghammer [19], pages 63–74.

[4] M. Angelaccio, T. Catarci, and G. Santucci. *QBD*: a graphical query language with recursion. *IEEE Transactions on Software Engineering*, 16(10):1150–1163, 1991.

[5] D. Bryce and R. Hull. SNAP: A graphics-based schema manager. In *Proceedings of the International Conference on Data Engineering*, pages 151–164, 1986.

[6] A. K. Chandra and D. Harel. Computable queries for relational databases. *J. Comput. Syst. Sci.*, 21:156–178, 1980.

[7] M. Consens and A. Mendelzon. GraphLog: a visual formalism for real life recursion. In ACM [1], pages 404–416.

[8] G. Engels. Elementary actions on an extended entity-relationship database. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science, International Workshop*, volume 532 of *Lecture Notes in Computer Science*, pages 344–362, Berlin, 1990. Springer-Verlag.

[9] R. Freund and B. Haberstroh. Attributed elementary programmed graph grammars. In Schmidt and Berghammer [19], pages 75–84.

[10] R. Freund, B. Haberstroh, and C. Stary. Applying graph grammars for task-oriented user interface development. In W. W. K. et al., editor, *Proceedings IEEE Conference on Computing and Information ICCI'92*, pages 361–365, May 1992.

[11] M. Gemis, J. Paredaens, and I. Thyssens. A visual database managment interface based on GOOD. In *Proceedings of the International Workshop on Interfaces to Database Systems*, 1992. To appear.

[12] K. Goldman, S. Goldman, P. Kanellakis, and S. Zdonik. ISIS: Interface for a Semantic Information System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, number 14:4 in SIGMOD Record, pages 328–342. ACM Press, 1985.

[13] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In ACM [1], pages 417–424.

[14] T. Houchin. Duo: Graph-based database graphical query expression. In Q. Chen, Y. Kambayashi, and R. Sacks-Davis, editors, *Proceedings of The Second Far-East Workshop on Future Database Systems*, volume 3 of *Advanced Database Research and Development Series*, pages 286–295, Singapore, Apr. 1992. World Scientific.

[15] C. Lewerentz and A. Schürr. GRAS, a managment system for graph-like documents. In *Proceedings Third Conference on Data and Knowledge Bases*, pages 19–31. Morgan Kaufmann Inc., 1988.

[16] M. Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierung.* Vieweg Verlag, 1979.

[17] J. Paredaens, J. Van den Bussche, M. Andries, M. Gemis, M. Gyssens, I. Thyssens, D. Van Gucht, V. Sarathy, and L. Saxton. An Overview of GOOD. *SIGMOD Record*, 21(1):49–53, 1992.

[18] P. Peelman, J. Paredaens, and L. Tanca. G-Log: A declarative graphical query language. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings 2nd International Conference on Deductive and Object-Oriented Databases*, number 566 in Lecture Notes in Computer Science, pages 108–128, Berlin, Dec. 1991. Springer-Verlag.

[19] G. Schmidt and R. Berghammer, editors. *Proceedings of the 17th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 570 of *Lecture Notes in Computer Science*, Berlin, 1992. Springer-Verlag.

[20] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating query languages for nearly-deterministic queries. In X. X, editor, *33nd Annual IEEE Conference on Foundations of Computer Science*. IEEE Computer Society Press, Oct. 1992.