Expressiveness and complexity of generic graph machines*

Marc Gemis Jan Paredaens
Peter Peelman Jan Van den Bussche†

University of Antwerp‡

Abstract

The Generic Graph Machine (GGM) model is a Turing machine-like model for expressing generic computations working directly on graph structures. In this paper, we present a number of observations concerning the expressiveness and complexity of GGMs. Our results comprise the following: (*i*) an intrinsic characterization of the pairs of graphs that are an input-output pair of some GGM; (*ii*) a comparison between GGM complexity and TM complexity; and (*iii*) a detailed discussion on the connections between the GGM model and other generic computation models considered in the literature, in particular the generic complexity classes of Abiteboul and Vianu, and the Database Method Schemes of Denninghoff and Vianu.

1 Introduction

Traditional models of computation, like the Turing Machine (TM) or the Random Access Machine (RAM), express computable functions mapping

*Work partially supported by the Belgian DPWB, program IT/IF/13.

†Current address: University of Limburg (LUC), Department WNI, B-3590 Diepenbeek, Belgium. E-mail: vdbuss@luc.ac.be.

‡UIA, Informatica, Universiteitsplein 1, B-2610 Antwerp, Belgium. E-mail: pareda@uia.ua.ac.be.

strings to strings or numbers to numbers. Sometimes, however, one is interested in computations that occur at a higher level of abstraction. For example, many information structures can be naturally represented as a labeled graph, and one is then interested in computable functions mapping graphs to graphs.

Of course, computations from graphs to graphs can be modeled on a TM, by encoding graphs as strings over some finite alphabet, or on a RAM, by encoding graphs as certain arrays of natural numbers. A fundamental problem with this approach, however, is that the encoding of a graph contains much more information than the graph itself. For example, in the TM encoding of a graph one can identify the node v_1 of the graph that comes first in the linear listing of all the nodes on the input tape; in the RAM encoding of a graph one can identify the node $v_{\max}$ that is maximal among all nodes, considering that the nodes are given as natural numbers. Nodes like v_1 or $v_{\max}$ thus have a special status in the encoding of a graph, while they may have no such status in the graph itself. (As an extreme example, in completely symmetric graphs like cliques or discrete graphs, no node is distinguishable from another node.)

Hence, when using conventional computation models like TMs or RAMs for expressing computations on graphs, one might want to restrict the computations to those that are "admissible", i.e., that do not depend on the extra information which is only there as an artifact of the encoding of the input. For example, a TM deleting the node v_1 (mentioned in the previous paragraph) from its input graph is not admissible. More precisely, a TM is admissible if the partial recursive function from strings to strings it computes, when decoded into a function from graphs to graphs, does not depend on the particular encoding of nodes as strings that is used, and neither on the particular ordering in which the nodes and edges are listed in the input string. (One can define a similar admissibility criterion for RAMs.)

The output graph of a graph function computed by an admissible computation device thus depends only on the input graph itself. This statement becomes even more clear if we consider the following intrinsic characterization of the admissible TMs: a graph function is computable by an admissible TM if and only if it preserves graph isomorphisms. (This characterization is not difficult to prove.) The formal property of preservation of isomorphisms, which is known as *genericity* [12], indeed captures nicely the informal property of depending only on the logical structure of the input graph.

A serious drawback of putting a genericity criterion on TMs however, is that genericity is not a syntactic notion (i.e., not recursively enumerable). What is therefore needed are computation models specifically tuned for expressing graph functions, working directly on the logical structure of the graphs so that genericity is guaranteed. One such generic computation model is the Generic Graph Machine (GGM), introduced by us in [10]. A configuration of a GGM consists of an underlying graph and a number of machine instances, each have a local state and pointing to two nodes of this graph. During the execution of a step, the machine instances perform in parallel a local transformation on the graph and are each replaced by number of other machine instances. Since the parallelism involved respects the symmetries of the graph, genericity is ensured.

The purpose of the present paper is to further our understanding of generic computation models by presenting a number of additional observations concerning the expressiveness and complexity of GGMs.

Many of our results relate the GGM model with data manipulation languages for object-oriented databases. Indeed, generic computation models have received considerable attention in database theory, in the form of database query and update languages. Genericity is particularly important in that context, since it captures the principle of physical data independence [4, 8, 12]. Initially, database languages, being used in the context of relational database systems, were designed for the computation of domain-preserving functions from relational structures to relations. But later, the focus shifted to the computation of graph functions, in the new context of object-oriented databases. Indeed, various manipulations of object databases can be modeled as graph manipulations [1, 5].

Concretely, we will analyze the relationship between the GGM model and the two object database graph languages GOOD [11] and DMS [9]. GOOD is a graph transformation language based on pattern matching, while DMS is an object-oriented language based on the parallel invokation of methods. We prove that DMS, a restricted version of GOOD, and a similarly restricted version of GGM, are equivalent to each other up to constant factors in time and space complexity, and equivalent to Turing machines up to polynomial factors. The interest in this result is that GOOD and DMS are not just aritrarily chosen languages. Indeed, GOOD is a yardstick for the computational completeness of formalisms for computing generic graph functions: GOOD programs are known to be able to compute precisely all "construc-

tive" generic graph functions [16]. Moreover, DMS is a yardstick for the complexity of such generic computations: it was designed to capture precisely the generic complexity classes introduced by Abiteboul and Vianu [3].

Our second main result is a direct proof of the "BP-completeness" of the GGM model. BP-completeness is an "intrinsic" property of the power of generic computation models, originally introduced in the context of query languages for relational databases by Chandra and Harel [8].[1] In the context of the GGM model it means that a pair (G, G') of graphs is an input-output pair of some GGM if and only if the group of automorphisms of G can be homomorphically embedded in the group of automorphisms of G', i.e., if the symmetries in G are preserved in G'. Andries and Paredaens [6] already proved the BP-completeness of GOOD; hence, the BP-completeness of the GGM model follows indirectly from the equivalence between GOOD and GGM mentioned above. The direct proof we will give is an "intrinsic" argument providing additional insight in GGM computations. We actually consider our argument to be clearer than the proof of BP-completeness of GOOD presented in [6].

This paper is further organized as follows. In Section 2, we recall the definition of the GGM model as introduced in [10]. In Section 3, we compare GGM complexity with TM complexity. In Section 4, we relate GGM complexity to generic complexity by giving mutual simulations between GGM, GOOD and DMS. Finally, in Section 5, we prove the BP-completeness of the GGM model.

2 Generic graph machines

A GGM, as illustrated in Figure 1, consists of a finite state control, a finite number of *machine instances* (MIs) and works on a labeled graph. Each MI is in some local state, and has a head and a pointer, both of which point to a node of the graph. Initially, the underlying graph is an arbitrary given input graph, and there is only one MI in a fixed starting state with head and pointer pointing to a particular fixed node. In Figure 1, the graph is enclosed in a rectangle, each MI is depicted as a gray circle labeled by its state, and the head and pointer of each MI are indicated by a dotted line.

[1] The BP is merely an abbreviation of Bancilhon-Paredaens [7, 14] and has nothing to do with NP or other complexity classes.

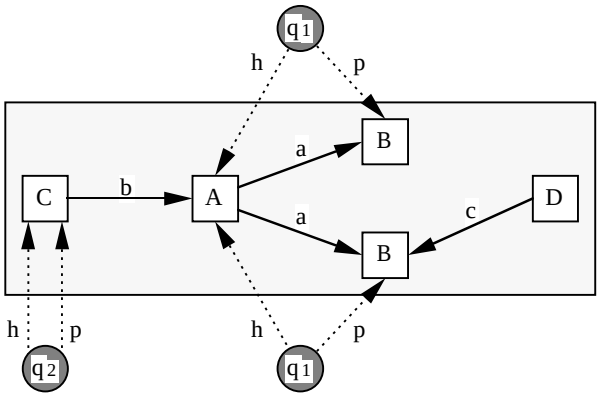

Figure 1: A GGM working on a graph. The MIs are depicted as circles.

In one transition of the GGM, each MI can—depending on its state—do the following:

1. Change state, or die;
2. Perform a simple action on the graph, such as the addition or deletion of a node or edge;
3. Move its head and pointer to other nodes, possibly splitting up into several new MIs.

The actions of the different MIs are performed in parallel.

To define the type of graph structures we will be working with, assume the existence of pairwise disjoint, infinitely enumerable sets **N** of *nodes*, **NL** of *node labels*, and **EL** of *edge labels*. We will generally denote node labels by capital letters and edge labels by lowercase letters.

Let NL and EL be finite subsets of **NL** and **EL**, respectively. A *graph over* (NL, EL) is a triple $G = (N, E, \lambda)$, where

- N is a finite subset of **N**;
- E is a subset of $N \times EL \times N$; and
- λ is a total function from N to NL.

For a node $n \in N$, $\lambda(n)$ is called the *label* of n. An element (n, a, m) of E is called an *edge from n to m with label a*. We will extend λ to E by putting $\lambda(n, a, m) := a$.

A *GGM* working on graphs over (NL, EL) now consists of

- a finite set of states, among which an initial state is designated;
- finite alphabets NL' and EL' of node and edge labels, respectively, containing NL and EL, respectively; nodes and edges created during an execution of the GGM will be labeled with elements from these alphabets; an initial node label in $NL' - NL$ is designated;
- a transition function from the set of states to a finite set of operations, to be specified shortly.

A *configuration* of a GGM as above is a pair (G, S), where G is a graph over (NL', EL'), and S is a set of triples of the form $(q, head, pointer)$, with $q \in Q$ and $head, pointer$ nodes in G. Each such triple is called a *machine-instance* (MI), consisting of a state q, a head placed on a node *head*, and a pointer placed on a node *pointer*.

We will define the transitions between configurations of a GGM in two steps. First, we define "elementary" transitions from configurations consisting of a single machine instance. Transitions from arbitrary configurations will then be parallel compositions of elementary ones.

Elementary transitions. Consider a configuration C of a GGM M consisting of one single MI only. This MI will transform C into another configuration C', denoted by $C \Longrightarrow_M C'$, according to the operation associated by M's transition function to the state the MI is in. The different possible operations are the following:

1. Die: the MI vanishes from the configuration.
2. Do nothing.
3. Test head and pointer for equality, changing to two different states depending on the outcome.
4. Move pointer to head.

5. Add an edge, with a specified label, from head to pointer, or from pointer to head.

6. Remove, if existing, the edge with a specified label from head to pointer, or from pointer to head.

7. Add a node, together with an edge to it from the head, with specified labels, and move head to the new node. A variant of this operation adds the edge in the reverse direction, i.e., from the new node to the head.

8. Delete the node under pointer, moving pointer to head; if head equals pointer die.

9. Look for nodes in the graph with some specified label; split into many MIs, one for each such node, with the head placed on that node. This is some kind of "global search". Alternatively, perform a "local search", looking for edges with some specified label going from the head to nodes with some specified label. In both cases, if no nodes are found, change to a different state.

In all situations listed above, the MI can, in addition to performing the operation, change to a new state. Note that global search is the only operation that is not "local", in the sense that it causes nodes of the graph to be visited that are not directly linked to the head or pointer of the MI. If global search were not allowed, an MI would never be able to leave a connected component; this would drastically diminish the computational power of GGMs (for example, it would no longer be possible to test whether there exist two nodes not linked by any edge).

General transitions. Now let $C = (G, S)$ be a general configuration of the GGM M, with $G = (N, E, \lambda)$, and let $I_1, \ldots, I_\ell$ be the MIs in S. For each $j = 1, \ldots, \ell$, let C_j be the configuration $(G, \{I_j\})$. With $C' = (G', S')$ another configuration of M, we say that *M transforms C into C' in one step*, denoted $C \Longrightarrow_M C'$, if the following conditions are satisfied:

1. There exist configurations $C'_1, \ldots, C'_\ell$ such that for each $j = 1, \ldots, \ell$, $C_j \Longrightarrow_M C'_j$. Let $C'_j = (G'_j, S'_j)$, with $G'_j = (N'_j, E'_j, \lambda'_j)$. For any two

different j and k, $(N'_j - N)$ and $(N'_k - N)$ must be disjoint; so the nodes added by the different MIs are different.

2. Writing $G' = (N', E', \lambda')$, we have:
   - $N' = \bigcap_{j=1}^{\ell} N'_j \cup \bigcup_{j=1}^{\ell} (N'_j - N)$;
   - $E' = \{(n, a, m) \in (\bigcap_{j=1}^{\ell} E'_j \cup \bigcup_{j=1}^{\ell} (E'_j \setminus E)) \mid n, m \in N'\}$; and
   - $\lambda' = \bigcup_{j=1}^{\ell} \lambda'_j$.
3. $S' = \{(q, head, pointer) \in \bigcup_{j=1}^{\ell} S'_j \mid head, pointer \in N'\}$.

The union defining λ' yields a well-defined function, since the different λ'_j agree on the intersections of their domains (these intersections will always consist of nodes in N).

It may be instructive to describe the formal definition of $\Longrightarrow_M$ intuitively as follows. In each step of the GGM, all MIs perform their operation on the graph in parallel. After each step, MIs whose head and pointer no longer belong to the graph are killed, and identical MIs merge into a single one. A GGM thus resembles a systolic automaton (see [13] for definition), except that the MIs are not fixed and permanent like individual systolic nodes. Various other computational models, which can be viewed as lying between GGMs and systolic automata, have been proposed in the literature. A selection of these has been reviewed in [10].

Duplicate Elimination. For the sake of clarity, in the preceding discussion we have left out one operation in the GGM model, which we now present separately. The *duplicate elimination* operation is not performed by individual MIs, but by the GGM as a whole. Informally, this operation replaces each equivalence class of "duplicate" nodes by a single new node serving as a unique representative for the class. To this end, we specify a number of states which we call "red". Each time some MI is in a red state, duplicate elimination if performed on the current configuration before moving to the next configuration.

Formally, let $C = (G, S)$ be a GGM configuration, and let n_1, n_2 be two nodes in G. We say that n_1 and n_2 are *duplicates with respect to* C if the transposition of n_1 and n_2, which can be applied to G and S in the canonical

way, leaves G and C invariant. Hence, n_1 and n_2 are logically interchangeable within the configuration.

The duplicate elimination now transforms configuration C into a new one $C' = (G', S')$ as follows. Call a node in G *red* if it is under the head of an MI in a red state. Let $\mathcal{Z}$ be the partition of the nodes in equivalence classes according to the following equivalence relation:

- If n_1, n_2 are red, then n_1 and n_2 are equivalent iff n_1 and n_2 are duplicates;
- If n is not red, then n is only equivalent to itself.

For each non-singleton equivalence class Z, we take a new representative node $n_Z \in U \setminus N$. We extend this to singleton equivalence classes $Z = \{n\}$ by putting $n_Z = n$. Now in the new configuration, each representative replaces its equivalence class. Formally, we have:

1. $G' = (N', E', \lambda')$, with
   - $N' = \bigcup_{Z \in \mathcal{Z}} \{n_Z\}$;
   - $\lambda'(n_Z) = \lambda(n)$ for $n \in Z$; and
   - $E' = \bigcup_{Z,Z' \in \mathcal{Z}} \{(n_Z, a, n_{Z'}) \mid \exists (n, a, m)$ in G, $n \in Z$, $m \in Z'\}$.
2. $S' = \{(q, n_Z, n_{Z'}) \mid \exists (q, n, m) \in S$, $n \in Z$, $m \in Z'\}$.

Computing graph functions. Now let C and C' be configurations of the GGM M, and let $\Longrightarrow_M$ be the transition relation between configurations of M as defined above. We say M *transforms* C *into* C', denoted $C \Longrightarrow_M^* C'$, if there exist configurations $C_1, \ldots, C_n$ such that

$$C \Longrightarrow_M C_1 \Longrightarrow_M \cdots \Longrightarrow_M C_n \Longrightarrow_M C',$$

and such that a node deleted in one of the transitions is not re-inserted in a later transition.

A *graph function* is a function mapping graphs to graphs. A GGM M computes a graph function as follows. Let A_0 be the initial node label of M, and let q_0 be the initial state. For a graph G, denote by $\bar{G}$ the graph obtained from G by adding a new node with label A_0. This new node is denoted by $n_{\bar{G}}$. We define:

Definition 2.1 M computes the graph function $\mathcal{F}$ if for each graph G on which $\mathcal{F}$ is defined,

$$(\bar{G}, \{(q_0, n_{\bar{G}}, n_{\bar{G}})\}) \Longrightarrow_M^* (\mathcal{F}(G), \emptyset).$$

So, when M is started on G with one single MI, in the initial state, and with head and pointer placed on an extra starting node, the final result graph when all MIs have died yields $\mathcal{F}(G)$. The extra starting node is necessary to deal with the case $G = \emptyset$; at the same time, it provides a simple and uniform way to start up the computation. We will say in this case that M *transforms* G *into* $\mathcal{F}(G)$.

Example. We conclude this section by showing how a GGM can test whether a tree is unbalanced, i.e., whether there are two paths from the root to a leaf with different length. We start with an MI having head and pointer on the root. This MI initiates a traversal of the tree, following every path by splitting in each internal node. Each MI that arrives in a leaf creates a node with some special fixed label not occurring in the tree, say B. Thus, if a MI arrives in a leaf and there is already a B-node, the tree is non-balanced, and the MI enters a special state to signal this.

3 Complexity

One can naturally define the *time complexity* of a GGM M as the maximum number of steps performed by any computation of M on a graph with n nodes, and the *space complexity* as the maximal number of nodes in any intermediate configuration of any computation of M on a graph with n nodes.

Since many different MIs work in parallel in one global GGM step, the time complexity of a GGM M alone does not say much about the classical Turing machine (TM) complexity of a graph function computed by M. This point is illustrated by the following example.

Example 3.1 The powerset of the set of nodes of a graph (an inherently exponential graph function) can be computed by a GGM with logarithmic time complexity as follows. We will create nodes labeled S, each linked by edges to the nodes of some subset of nodes of the input graph. We start by creating nodes for the empty set and the singleton sets. We then enter a

loop at each iteration of which we create for every pair of S-nodes a new one linked to the union of their associated sets. This is repeated as long as an S-node linked to all nodes is not yet created. Since after i iterations all sets containing at most 2^i elements are represented, the time complexity of the computation is $O(\log n)$.

We can nevertheless give a (rough) upper bound on the number of nodes which can be created by a GGM in a polynomial number of steps, as follows:

Proposition 3.2 *No GGM can create more than $O(f(n))$ new nodes in a polynomial number of steps, where*

$$f(n) = 2^{2^{2^n}} \cdot n^{2^{2^n+1}}.$$

Proof. The only operation which can create new nodes is the node addition operation. The maximal number of new nodes that can be created in one step equals the number of MIs in the configuration of that step. Let $N(i)$ ($M(i)$) be the number of nodes (MIs) in the configuration after i steps. Note that $N(0) = n$ and $M(0) = 1$. We thus have

$$N(i+1) \leq N(i) + M(i).$$

Moreover, by definition of MI, we have

$$M(i+1) \leq q(N(i+1))^2,$$

where q is the number of states of the GGM. Solving for these two inequalities yields

$$N(i) \leq 2^{2^i-1} \cdot q^{2^{i-1}-1} \cdot n^{2^{i-1}}.$$

For sufficiently large $n \leq q$, and bounding the number of steps i from above by 2^n, we obtain the desired result. ■

The preceding discussion motivates us to look at the *combined* space-time complexity of a GGM computation as a more appropriate measure of its "cost". Indeed, under this view we can show the following relationship between GGM complexity and TM complexity:

Proposition 3.3 *A GGM with time complexity $T(n)$ and space complexity $S(n)$ can be simulated by a TM with time complexity polynomial in $T(n) \cdot S(n)$.*

Proof. We will use a 7-tape TM. Tape 1 is used for the generation of the new node identifiers. Its maximum length is $O(\log(S(n)T(n)))$. Tape 2 holds the node identifiers with their label. It contains a sequence of elements of the form (i, A, t), with i a node identifier, A a label and t a tag that can have a finite number of possible values and that is used during the computation. The maximum length of Tape 2 is $O(S(n)\log(S(n)T(n)))$. Tape 3 holds the new node identifiers created during the current step. It has the same form and maximum length as Tape 2. Tape 4 holds the edges with their label. It contains a sequence of elements of the form (i, a, j, t), with i and j node identifiers, a an edge label and t a tag. The maximum length of Tape 4 is $O(S(n)^2\log(S(n)T(n)))$. Tape 5 holds the new edges created during the current step. It has the same form and maximum length as Tape 4. Tape 6 holds the MIs. It contains a sequence of elements of the form (q, i, j, t), with i and j node identifiers, q a state and t a tag. The maximum length of Tape 6 is also $O(S(n)^2\log(S(n)T(n)))$. Tape 7, finally, holds the MIs after the current step and has the same form and maximal length as Tape 6.

The Turing machine simulates each step of the GGM by first performing a duplicate elimination (if in a "red" state) and then running through Tape 6, and simulating each MI. The latter part is done in $O(S(n)^2F(n))$ steps ($F(n)$ will be explained later). Afterwards Tapes 2 and 3 are adjusted (taking $O(S(n)\log(S(n)T(n)))$ steps), Tapes 4 and 5 are adjusted (taking $O(S(n)^4\log(S(n)T(n))^2)$ steps) and Tapes 6 and 7 are adjusted (also taking $O(S(n)^4\log(S(n)T(n))^2)$ steps).

A duplicate elimination takes $O(S(n)^6\log S(n)T(n)^4)$ steps. For each pair of nodes, one has to test whether they are duplicates. The number of steps needed to find all pairs of nodes is $O(S(n)^2\log S(n)T(n)^2)$. For each pair one has to compare the set of outgoing edges; this takes $O(S(n)^4\log S(n)T(n)^2)$ steps.

The value of $F(n)$ depends on the kind of step performed. The local search operation is the most time-consuming case to simulate. This operation requires scanning Tape 4, to find an edge starting in the given head and with label a, and then scanning Tape 2 to check whether the end node has label A. This takes $O(S(n)^3\log S(n)T(n)^2)$. So, in total, the simulation of the effect of the MIs takes $O(S(n)^5\log S(n)T(n)^2)$ steps.

Of the three parts, the simulation of the duplicate elimination is the most costly. So in general the time complexity is $O(S(n)^6.\log S(n).T(n)^4)$. ■

A converse to the above simulation result is easily seen:

Proposition 3.4 *Each TM can be simulated in real time by a GGM.*

Proof. We encode the tape of the TM by a linked list of cell nodes. The cells are labeled by the tape symbol they contain. Initially, only the part of the tape that contains the input (i.e., the part that is non-blank) is represented. If more of the tape is needed, new cell nodes are created dynamically. It is now straightforward to simulate the configuration transitions of the TM. ■

It may be interesting to note that in the above simulation, the global search operation is not needed.

4 Generic computation models

As already mentioned in the Introduction, one of the main goals of the GGM model is its *genericity*. This means that the computation of a GGM working on a graph happens completely on the logical level. In other words, the computation depends exclusively on the logical structure of the graph, not on the way the nodes and edges of a graph are "physically" represented and stored in the computer. Conventional computation models like the RAM or the Turing machine are not generic (they were of course not meant to be). For example, a computation of a RAM on a graph can exploit the fact that the nodes of the graph are really natural numbers. For another example, when providing a Turing machine with a graph as input, one actually gives the graph *plus* an ordering of its nodes (induced by the ordering of the input tape cells); this ordering is strictly not part of the graph's logical structure.

4.1 Generic completeness

A natural question to ask is whether the GGM model is computationally complete in some natural sense. The notion of completeness with respect to generic computations has been addressed extensively in the literature on the theory of database queries (e.g., [2, 8]). Generic computations are particularly relevant to database applications, since genericity is intimately connected to the fundamental notion of *data independence* [4]. Traditionally, the

theory of database queries has focused on the computation of generic functions on relational structures in general rather than graph structures in particular. Moreover, attention is traditionally restricted to *domain-preserving* functions; as such, the theory is not fully applicable to the particular case of graphs, as domain-preserving graph functions would not be able to add new nodes to a graph.

However, motivated by recent applications of databases in object-oriented environments, the theory of database queries has recently been extended beyond domain-preserving functions, allowing for the introduction of new domain elements in the result of a query [1, 16]. This extended theory, needed to account for the creation of new objects in an object-oriented database, is fully applicable to the particular case of graphs. A powerful language for expressing object-creating queries is the language provided by the GOOD model, a graph-oriented model for object databases [11]. It has been shown that the GOOD language is computationally complete in that the language can express precisely all *constructive* computable generic graph functions [15, 16].

Hence, to assess the completeness of the GGM model, its expressive power must be compared to that of the GOOD language. We have done this in a previous paper [10], and were indeed able to prove that GGM is equivalent to GOOD, thereby establishing its completeness. In order to recall this result in the present paper, we first briefly introduce the GOOD language itself, in a simplified form sufficient for our purposes.

GOOD is a procedural programming language for working with graphs. It provides five types of basic statements and is closed under composition and while-loops in the usual way. The five types of basic statements are the following:

1. *Edge addition.* Let G be a graph, P another, fixed, graph (called the *pattern*), let n, m be (not necessarily distinct) nodes in P, and let e be an edge label. Applying the edge addition operation

$$\mathrm{EA}[P, n, m, e]$$

to G yields a graph G', obtained from G by adding, for each homomorphism f from P into G (called a *matching* of the pattern P in G), the edge $(f(n), e, f(m))$.

It is important to note that the pattern is a fixed part of the "syntax" of the operation; different patterns give rise to different edge addition operations. As a consequence, an edge addition is polynomial-time computable. The same holds for the remaining four operations.

2. *Node addition.* Let $n_1, \ldots, n_k$ be nodes in P, let $e_1, \ldots, e_k$ be edge labels, and let K be a node label. Applying the node addition operation

$$\mathrm{NA}[P, n_1, \ldots, n_k, e_1, \ldots, e_k, K]$$

to G yields a graph G', obtained from G by adding, for each matching f of P in G, a K-labeled new node with an e_i-labeled edge to $f(n_i)$ for each $i = 1, \ldots, k$, provided such a node does not already exist in G'.

3. *Edge deletion.* Let (n, e, m) be an edge in P. Applying the edge deletion operation

$$\mathrm{ED}[P, n, e, m]$$

to G yields a graph G', obtained from G by deleting, for each matching f of P in G, the edge $(f(n), e, f(m))$.

4. *Node deletion.* Let n be a node in P. Applying the node deletion operation

$$\mathrm{ND}[P, n]$$

to G yields a graph G', obtained from G by deleting, for each matching f of P in G, the node $f(n)$.

5. *Abstraction.* This operation is different in spirit from the other four and is used for duplicate elimination. Let n be a node in P, let e, e' be edge labels, and let K be a node label. We can define the following equivalence relation among the nodes of G:

$$m \equiv_e m' \Leftrightarrow \{m'' \mid \text{edge } (m, e, m'') \text{ in } G\} = \{m'' \mid \text{edge } (m', e, m'') \text{ in } G\}.$$

We say in this case that m and m' are *duplicates w.r.t. e*. Denote the restriction of $\equiv_e$ to those nodes that equal $f(n)$ for some matching f of P in G by $\equiv_e |_P$. Then applying the abstraction operation

$$\mathrm{AB}[P, n, e, e', K]$$

to G yields a graph G', obtained from G by adding, for each equivalence class Z w.r.t. $\equiv_e |_P$, a K-labeled new node with an e'-labeled edge to each $m \in Z$, provided such a node does not already exist in G'.

GOOD programs can now be defined as follows. Each basic statement is a program. If Q_1 and Q_2 are programs, then their *composition* $Q_1; Q_2$ is a program. Finally, if Q is a program and K is a node label, then the *while-loop* **while** K **do** Q **od** is a program. The semantics of basic statements was defined above. The semantics of a composition $Q_1; Q_2$ is the obvious one: if G is a graph, G' is a graph obtained from applying Q_1 to G, and G'' is a graph obtained from applying Q_2 to G', then applying $Q_1; Q_2$ to G yields G'', on condition that a node which is deleted by a node deletion operation in Q_1 is not added back by a node addition or abstraction operation in Q_2. The semantics of a while-loop **while** K **do** Q **od** is "repeat Q as long as there is a K-labeled node in the graph."

4.2 Generic complexity

There is often a serious mismatch between the conventional Turing complexity of a computational task and its complexity when performed in a generic computation model. This phenomenon was studied in detail by Abiteboul and Vianu [3] in the context of traditional domain-preserving relational queries. They introduced *generic*, rather than classical Turing machine, complexity classes, based on a generic computation model, called the *generic machine (GM)*. The GM model is an adaptation of the basic Turing machine model to compute generic functions on relational structures. A striking illustration of the difference between classical and generic complexity classes is provided by the problem of parity checking: Abiteboul and Vianu proved that the simple computational problem of determining whether the total number of elements in a given structure is even, is not in generic PSPACE.

Although GMs and the GGMs of the present paper, both being geared towards generic computation, are very similar in spirit, a combination of formal differences makes it awkward to compare them directly. It would therefore be useful if instead we could compare the GGM model with an alternative model which yields essentially the same generic complexity classes but is designed for computing object-creating graph functions. Fortunately, such an alternative model already exists: the *database method scheme (DMS)*

model proposed by Denninghoff and Vianu [9]. We will now briefly introduce this graph-based model, in a simplified form sufficient for the purpose of the present paper.

In DMS, a graph is viewed as an object-oriented database: nodes are viewed as database objects and edges between nodes are viewed as attribute relationships between objects. Attention is restricted to graphs with *single-valued edge labels*; with this we mean that for each edge label e, there are never two different edges labeled e leaving the same node. A DMS program consists of a number of *methods*: straight-line procedures which are run local to some database object (called the *receiver* of the method and denoted by the word *self*). Throughout the computation, several method invocations are active, and they run synchronously in parallel. A method body has a number of *variables* $x_1, \ldots, x_k$ holding nodes as values, some of which can be initialized as parameters, and consists of a sequence of *statements*, each of one of the following types:

1. A *variable assignment* of the form $x_i := r$, where r is either *(i)* an edge label e; *(ii) self*; or *(iii)* a variable. In case *(i)*, x_i is assigned the target node of the edge labeled e leaving *self*, if existing; cases *(ii)* and *(iii)* have the obvious semantics.

2. An *edge addition* of the form $e := x_i$, where e is an edge label. An edge labeled e is added between *self* and x_i.

3. An *edge deletion* of the form **delete** e. The edge labeled e leaving *self* is removed.

4. A *node addition* of the form $x_i := \mathbf{new} : C(e_1 : y_1, \ldots, e_m : y_m)$, where C is a node label, the e_j are edge labels, and the y_j are variables. A new node n with label C is added and assigned to x_i, and for every $j = 1, \ldots, m$ an edge labeled e_j is added from n to y_j. If several method invocations execute a node addition in parallel, a distinct node is created for each invocation.

   There is also a variant of the node addition, denoted by $\mathbf{new}^{\mathrm{val}}$. In this variant, if several method invocations in parallel execute a node addition with exactly the same parameters, *at most one* new node is created, depending on whether or not there is already a node n with

label C and for each j e_j-labeled edges to y_j in the graph. In other words, no "duplicate" nodes are created.

5. A *method invocation* of the form **send** $M : C(y_1, \ldots, y_p)$, where M is the name of a method, C is a node label, and the y_j are variables. For each node n with label C, a new parallel process is started, executing method M on receiver n with parameters $y_1, \ldots, y_p$.

6. A *node deletion*: **delete** *self*. This deletes the node *self* and the method invocation executing this statement comes to an end.

7. Finally, a statement of the form **if** *condition* **then** *statement*, where *condition* is a boolean combination of elementary conditions of the form $x_i = x_j$ or $x_i \neq x_j$, and *statement* is not another **if** statement, with the obvious semantics.

The DMS program is started by an external invocation **send** $M : C$ for some node label C and some method M without parameters. As already mentioned, the methods run synchronously in parallel; at every global step of the computation, every active method process executes one statement.[2] Whenever there are conflicting parallel statement executions (e.g., an edge addition and an edge deletion of the same edge, or two conflicting parallel edge additions which would violate the single-valued edge label property of the graph) the global result of the computation becomes undefined (the computation "crashes").

DMS programs compute graph functions in the obvious manner, and DMS computations are also clearly generic. Our main interest for the DMS model lies in the generic complexity classes that can be defined in the context of this model [9]. Let DMS-PSPACE be the family of graph functions computable by a DMS program which, at each point in its computation on an input graph G, uses a number of nodes and method processes that is at most polynomial in the size of G. Let DMS-PTIME be the family of graph functions in DMS-PSPACE computable by a DMS program whose computation on an input graph G performs a number of global steps at most polynomial in the size of G. Note that the requirement that DMS-PTIME functions must be in

[2]Statements of the form **if** *condition* **then** *statement* form the only exception: here, *condition* is evaluated in one step and, depending on the result of this evaluation, either the *statement* or the next statement in the sequence is executed in the following step.

DMS-PSPACE is necessary, since DMS programs can generate a result of exponential size in a polynomial number of steps, just as is the case for GGMs as we saw in Section 3.

Denninghoff and Vianu [9] argued convincingly that the DMS-complexity classes are the natural extensions of the aforementioned generic complexity classes of Abiteboul and Vianu, from the domain-preserving relational queries to the object-creating generic graph functions.

4.3 The robustness of generic complexity for graph functions

Note that we now have three formalisms for expressing graph functions: GGM, GOOD, and DMS. Analogously to the way DMS-PSPACE and DMS-PTIME have just been defined, one can define GGM-PSPACE and GGM-PTIME. Namely, GGM-PSPACE is the family of graph functions computable by a GGM which, at each point in its computation on an input graph G, has a configuration of size at most polynomial in the size of G. GGM-PTIME then is the family of graph functions in GGM-PSPACE computable by a GGM whose computation on an input graph G consists of a number of transitions at most polynomial in the size of G. And we can also define GOOD-PSPACE as the family of graph functions computable by a GOOD program for which, at each point in its computation on an input graph G, the intermediate result graph has size at most polynomial in the size of G. GOOD-PTIME then is the family of graph functions in GOOD-PSPACE computable by a GOOD program whose computation on an input graph G executes a number of basic statements at most polynomial in the size of G.

The basic DMS model as defined in [9] is restricted to single-valued edges and has no duplicate elimination capability like that of GGMs. *Correspondingly, for the remainder of this section we restrict attention to GGMs without duplication elimination, to GOOD programs without the abstraction operation, and to graphs with single-valued edge labels.* Under this restriction we are going to establish the following result, evidencing the naturalness and robustness of the proposed generic complexity classes for graph functions:

Theorem 4.1 *GGM-PSPACE = GOOD-PSPACE = DMS-PSPACE, and GGM-PTIME = GOOD-PTIME = DMS-PTIME.*

Proof. One step of a GGM, a GOOD program, or a DMS program can increase the size of the current configuration by at most a polynomial. The theorem is therefore proven if we can establish mutual simulations in "lock-step", meaning that for each machine or program Q in one model there exist a machine or program Q' in the other models such that for some constant c, every computation of Q consisting of n steps is simulated by Q' in $c \cdot n$ steps.

Lockstep simulations from GOOD to GGM and vice versa are already known from our previous work [10]. To prove the theorem, we furthermore establish lockstep simulations from DMS to GOOD and from GGM to DMS.

GOOD can simulate DMS in lockstep. We describe for a given DMS program Q a GOOD program Q' that simulates Q. It will be clear that the simulation has only a constant step-overhead.

An active method process is simulated by a node labeled *Method*, with an edge labeled *self* to the receiver node of the process. The process nodes are tagged with a loop-edge labeled by the method name. The values of the variables of a process are indicated by edges, labeled by variable names. A process node is also tagged with a loop-edge labeled by the statement number to be executed. Assuming the DMS program Q is started up with the invocation **send** $M : C$, we thus have the following overall structure for GOOD program Q':

NA[$P, n,$ *self*, *Method*];
(where P is the pattern with one node n labeled C)
EA[P, n, n, M];
(where P is the pattern with one node n labeled *Method*)
EA[$P, n, n, 1$];
(where P is as in the previous statement)
while *Method* **do**
Simulate a global step of Q;
od

The simulation of a global step of Q in the body of the above loop begins by tagging each process node with a loop-edge labeled *one-step* (using an edge addition operation). Then follows, for each combination (M, i) where M is a method name of Q and i is the sequence number of some statement in the body of M, a group of GOOD statements. These GOOD statements perform three tasks. First comes the actual simulation of the parallel execution of the

i-th statement of M by all method processes which have name M and which must indeed execute the i-th statement. Denote the set of these processes by $active(M, i)$. Second, the $active(M, i)$ nodes are un-tagged by deleting the *one-step* loop-edges. Third, the process node is tagged with the next statement number, if there is one; if not, the process node is deleted. We will focus on the first task; the second and third tasks are straightforward applications of edge deletion, edge addition and node deletion.

The process nodes of $active(M, i)$ can be detected by using GOOD patterns containing a node labeled *Method* which is tagged three loop-edges: one labeled M, the second labeled i, and the third labeled *one-step*. We now consider the different possibilities for the i-th statement. We will no longer spell out the GOOD constructions in detail; the reader may wish to consult [11] for various examples on how to program in GOOD.

- The simulation of variable assignment, edge addition and edge deletion statements in DMS amounts to straightforward applications of the edge deletion and edge addition operations of GOOD.

- The simulation of a DMS node addition statement is performed by a GOOD node addition operation. If the basic **new** is used, the node addition operation must specify the created node to be connected to the method process node. If the $\mathbf{new}^{\mathrm{val}}$ variant is used, this connection is omitted.

- DMS's method invocation is also simulated using a GOOD node addition operation which creates the new process nodes.

- DMS's node deletion is simulated using two GOOD node deletions: one to delete the receiver node of the method, the other to terminate the process.

- Finally, a DMS statement **if** *condition* **then** *statement* is simulated by giving it two sequence numbers: one for the *condition*, and the other for the *statement*. The evaluation of the *condition* is possible in GOOD (details omitted).

DMS can simulate GGM in lockstep. We only sketch the simulation. GGM machine instances (MIs) are simulated using DMS method processes.

The head and pointer of an MI are simulated using local variables. The simulation of the elementary actions performed by the MIs (head-pointer assignments, edge additions and deletions, node creations, pointer deletions) is straightforward. GGMs use flowchart-like program control (implicit in the state-transition function δ), while the control flow of DMS programs is expressed using procedure calls; it is well-known how the former mechanism can be simulated by the latter. A global search operation looking for A-labeled nodes is simulated by a method invocation of the form **send** $M : A(self)$. A local search operation looking for A-labeled nodes linked to the head by an a-labeled edge is simulated similarly, but now M must check whether the desired a-edge is present. ■

5 BP-completeness

Up to now in this paper, we have investigated the expressiveness and complexity of GGMs computing global graph functions. In this final section, we characterize the expressiveness of the GGM model in terms of individual transformations, i.e., of pairs of input-output graphs. The same characterization in the context of the GOOD model (introduced in the previous section) is already known [6]. Our contribution in what follows is to provide a direct proof in terms of the GGM model, so as to further our understanding of generic computation models.

We need the following notation and terminology. For a graph G, $Aut(G)$ denotes the set of automorphisms of G.[3] This set forms a group under composition. For two graphs G and G', an *extension morphism from G to G'* is a group homomorphism $h : Aut(G) \rightarrow Aut(G')$ such that for each $f \in Aut(G)$, f and $h(f)$ coincide on $G \cap G'$.

We have:

Theorem 5.1 *The following are equivalent:*

1. *There exists a GGM which transforms G into G';*
2. *There exists an extension morphism from G to G'.*

[3] An automorphism is a permutation of the set of nodes which leaves the graph invariant, i.e., preserves edges and labels.

Proof. For the implication from (1) to (2), let $G \Longrightarrow_M^* G' \Longrightarrow_M G''$ for a GGM M, and assume as inductive hypothesis that an extension morphism h' from G to G' exists. We must show that an extension morphism h'' exists from G to G''. This is readily verified by case analysis on the different operations that can be performed by MIs in the transition from G' to G'', adapting h' into the needed h''. For instance, if a new node n is created by an MI (q, n_1, n_2) executing a node addition operation, then by symmetry, for any $f \in Aut(G)$ the MI $(q, h'(f)(n_1), h'(f)(n_2))$ will also create a new node m, and we define $h''(f)(n) := m$.

The implication from (2) to (1) is the more difficult one to prove. Assume an extension morphism h exists from G to G'. We have to show that G can be transformed into G' by some GGM. Actually, we will use an "extended GGM" as a technical convenience for this proof. In an extended GGM, the MIs do not consist of just one head and one pointer, but instead they can keep an unbounded array of nodes in their private memory. The extended GGM operations are like the normal GGM operations, except that they are indexed by array positions, indicating the nodes in the memory of an MI executing an operation that will be involved in the operation. The reader is invited to verify that extended GGMs are no more powerful than ordinary GGMs: the former can be simulated using the latter in lockstep (the array of nodes is simulated by a linked list of auxiliary nodes serving as pointers to the array elements).

The algorithm executed by the extended GGM is the following:

1. *Create an MI for every automorphism of G.* First, we create one MI for each permutation of the nodes of G that preserves node labels. The permutations are stored in the arrays of the MIs as listings of the nodes of G. Thereto, fix one arbitrary listing $n_1, n_2, \ldots$ of all nodes in G. Assume A_i is the label of n_i. Then for each $i = 1, 2, \ldots$ in succession we look for all nodes with label A_i, splitting each current MI into one new MI for each node labeled A_i, which is appended to the array. After these steps we select those MIs whose arrays contain no duplicate elements. This yields the needed permutations. Finally, to obtain the automorphisms of G, we select from these permutations those for which there exists an a-labeled edge in the graph between the i-th and j-th node in the array, whenever this is the case for n_i and n_j.

2. At this point, the i-th node in the array of the MI representing the

automorphism f equals $f(n_i)$. If n_i is also in G' then we know that $f(n_i) = h(f)(n_i)$.

3. *Add all edges in G' not in G between nodes in the current configuration.* (The first time this step is executed, the current configuration consists of G, but later new nodes will be added.) Whenever an edge exists in G' between n_i and n_j, we let each MI add such an edge between the i-th and j-th node in its array. Note that this will not only add the edge between n_i and n_j, but also between $h(f)(n_i)$ and $h(f)(n_j)$ for each automorphism f of G and some extension morphism h from G to G'. But we know these edges are also present in G' since $h(f)$ is an automorphism of G'.

4. *Choose an arbitrary node n in G' not yet in the current configuration, add it together with certain other nodes, as well as all edges in G' linking these added nodes to nodes in the current configuration.* Consider the set $O = \{h(f)(n) \mid f \in Aut(G)\}$. If the nodes in the current configuration are listed as $n_1, \ldots, n_k$, list the nodes in O in some arbitrary but fixed order as $n_{k+1}, \ldots, n_{k+\ell}$. Note that not only n, but *all* nodes in O are not in G. If the label of n (and all other nodes in O) is A, let each MI add a new A-node (appending it to its array) ℓ times. Next, add the edges in G' between some node n_i with $1 \leq i \leq k$ and some node n_j with $k+1 \leq j \leq k+\ell$ as in the previous Step 3, by letting each MI add an edge between the i-th and j-th node in its array. Finally, perform a duplicate elimination on the nodes occurring in array positions $k+1, \ldots, k+\ell$ of the MIs.

   The net effect of the above operations is that precisely the nodes in O have been added to the current configuration together with the edges in G' linking them to nodes in the current configuration. We can see this formally by identifying the ℓ nodes added by the MI representing the identity automorphism with the nodes $n_{k+1}, \ldots, n_{k+\ell}$ of O. We must then verify that the duplicate elimination has the desired effect, i.e., we must prove two things:

   (a) Each node added by some MI is a duplicate of some element of O;

   (b) No two different elements of O are duplicates.

The last item is clear, since the elements of O occur in some fixed order and thus cannot be swapped. For the first item, assume m is added as the j-th node added by the MI representing the automorphism f. Then m is a duplicate of $h(f)(n_{k+j})$. Indeed, an edge $(x, e, h(f)(n_{k+j}))$ is present if and only if the edge $(h(f^{-1})(x), e, n_{k+j})$ is, and this edge in turn is added by the MI representing the identity automorphism if and only the edge (x, e, m) is added by the MI representing f.

5. At this point, the MI representing the identity has in its array a partial listing $n_1, \ldots, n_{k+\ell}$ of the nodes of G'. Each other MI, representing an automorphism f of G, has in its array the listing $h(f)(n_1), \ldots, h(f)(n_{k+\ell})$. We can now repeat the previous Steps 3 and 4 until all nodes and edges of G' have been added.

6. Finally, we need to remove the nodes and edges in G not in G'. For each such node n_i we let each MI delete the i-th node in its array, and for each such edge (n_i, e, n_j) we let each MI delete the e-edge between the i-th node and the j-th node in its array. This will not only remove the originally chosen nodes and edges but also their images under all automorphisms of G. But we know that these images and edges are not in G' either because there is an extension morphism from G to G'.

■

References

[1] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, volume 18:2 of *SIGMOD Record*, pages 159–173. ACM Press, 1989.

[2] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2):181–229, 1990.

[3] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings 23rd ACM Symposium on Theory of Computing*, pages 209–219, 1991.

[4] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[5] 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, *Advances in Database Technology—EDBT'92*, volume 580 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 1992.

[6] M. Andries and J. Paredaens. On instance-completeness of database query languages involving object creation. *Journal of Computer and System Sciences*, 52(2):357–373, 1996.

[7] F. Bancilhon. On the completeness of query languages for relational data bases. In *Proceedings 7th Symposium on Mathematical Foundations of Computer Science*, volume 64 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1978.

[8] A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[9] K. Denninghoff and V. Vianu. The power of methods with parallel semantics. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 221–232, 1991.

[10] M. Gemis, P. Peelman, J. Paredaens, and J. Van den Bussche. A computational model for generic graph functions. In H.J. Schneider and H. Ehrig, editors, *Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 170–187. Springer-Verlag, 1994.

[11] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, August 1994.

[12] R. Hull and C.K. Yap. The format model, a theory of database organization. *Journal of the ACM*, 31(3):518–537, 1984.

[13] H.T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982.

[14] J. Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7(2):107–111, 1978.

[15] J. Van den Bussche. *Formal aspects of object identity in database manipulation*. Doctor's thesis, University of Antwerp (UIA), 1993.

[16] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319, 1997.