PPEM-10

Graph Grammar Based Specification of Interconnection Structures for Massively Parallel Computation*

Duane A. Bailey
Janice E. Cuny

COINS Technical Report 87-23
June 1987

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003
USA

*The Parallel Programming Environments Project at the University of Massachusetts is supported by the Office of Naval Research, contract N000014-84-K-0647. Duane Bailey was also supported by an American Electronics Association ComputerVision fellowship.

Abstract

Algorithms designed for highly parallel processing often require specific interprocess communication topologies, including vectors, meshes, trees, toruses and cube-connected structures. Static *communication structures* are naturally expressed as graphs with regular properties, but this level of abstraction is not supported in current environments. Our approach to programming massively parallel processors involves a *graph editor*, which allows the programmer to specify communication structures graphically. As a foundation for graph editor operations, we are currently investigating properties of *aggregate rewriting graph grammars* which rewrite, in parallel, aggregates of nodes whose labels are logically related. We have found these grammars to be efficient in their description of many recursively defined graphs. Languages generated by these grammars can be associated with *families* of graphs. We also suggest extensions to the formalism that make use of extended labeling information that would be available in graph editors.

1. Introduction

It will soon be possible to experiment with the first massively parallel processors - ensembles of a thousand or more individually programmed computing elements. Algorithms for these machines are often characterized by relatively small, tightly coupled processes that require extensive interprocess communication. The communication channels needed for an algorithm collectively form its *communication structure*. These structures are fundamental to massively parallel computation: they form the basis for our understanding of algorithms, and the characteristics of their implementation often dominate program performance. Furthermore, the extent of their specification affects the feasibility of automatic error detection and correction. It is surprising, then, that few of the programming environments and languages proposed for these new machines provide for the explicit specification of communication structures.

Programming environments that do provide for explicit specification of communication structures use one of two approaches: either the structures are specified textually[1,8] or they are specified graphically[10]. Graphical specification is the more natural; parallel algorithms found in the literature are often accompanied by a graph. Existing tools for specifying these graphs, however, are inadequate for two reasons. Firstly, they rely on the user to manually draw interconnections, which is obviously infeasible for large numbers of processors. Secondly, they ignore the fact that the number of required processors (and hence the graph size) is often dependent on problem size; thus the programmer wants to specify not graphs but graph families.

A *graph editor* based on a parallel graph rewriting mechanism could provide an efficient tool for the specification of families of communication structures. In addition, by using such an editor to describe both logical and physical structures, it may be possible to generate or control mapping strategies by comparing graph constructions. As a foundation for graph editor operations, we report here on a graph rewriting mechanism that models methods that programmers use in constructing graphs. We also address variations in this mechanism that make use of labeling information that would be available in applications, such as communication graph editors.

In the next section, we discuss characteristics of communication structures in highly parallel environments. Section 3 presents the formalism of *aggregate rewriting graph grammars* and examples of grammars that concisely describe large, 'regular' graphs. Section 4 examines possible extensions to the labeling mechanisms of these grammars.

2. Characteristics of Communication Structures

Like any editor, a graph editor for communication structures should be biased towards correct constructions - in this case, of graphs commonly used in parallel algorithms. Typically, these graphs are 'regular' in the sense that they have one or more of the following characteristics (Figure 1):

- *Sparsity.* Most communication structures are sparse: for a family of graphs the ratio of channels to processes is roughly constant. This occurs because physical constraints limit the number of direct connections to each processor.
- *Recursive Construction.* Most algorithms are designed in the small but are intended for arbitrarily large machines. Often this results in scalable, recursive structures such as trees, cubes, and multistage permuting networks.
- *Near Symmetry.* Most parallel algorithms are characterized by a number of identical processes, each having the same 'local' view of the communication structure. Some of the resulting structures, such as the n-cube, are completely symmetrical but others, such as grids, have 'edge effects', that is, minor differences in programming due to input/output requirements at the boundaries of the process structure.
- *Low radius.* Most algorithms require some nonlocal communication which is more efficiently realized in structures with low radius.

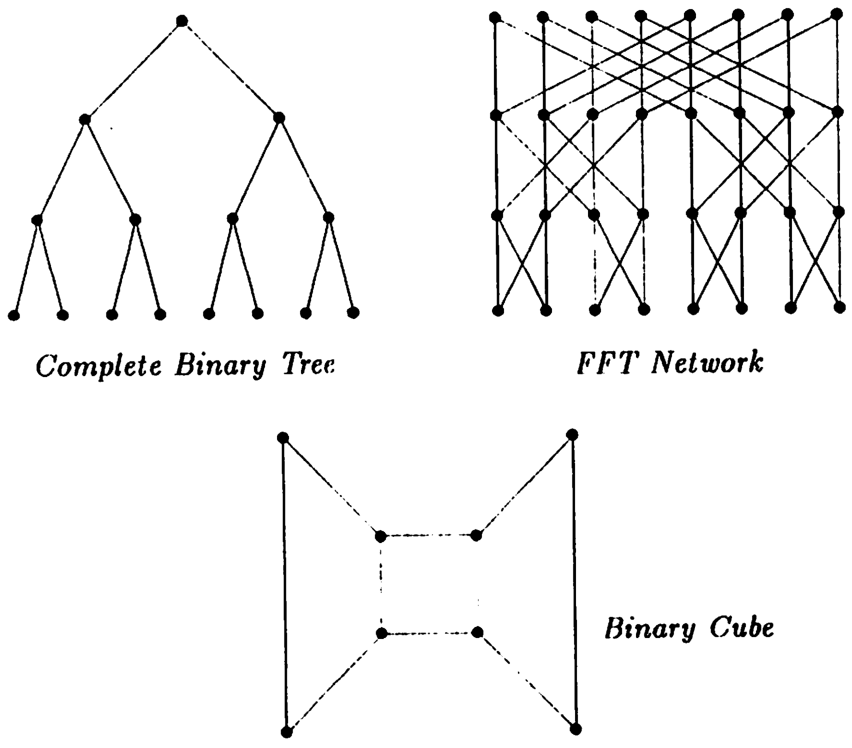


Figure 1: Three common communication structures. The tree is typical of a divide-and-conquer algorithm, the FFT network models polynomial evaluation, and the binary cube is typical of physical modeling programs.

In pursuing efficient graph editor operations, therefore, it is sensible to identify mechanisms that are capable of constructing large 'regular' structures in a directed manner.

3. Aggregate Rewriting Graph Grammars

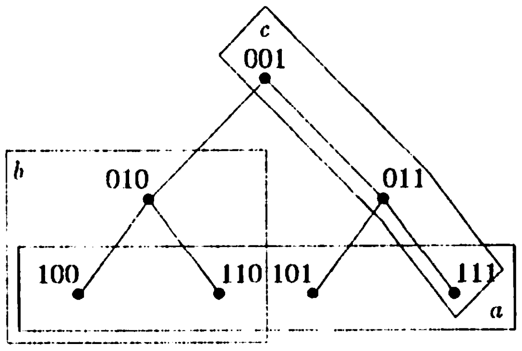


Figure 2: Graph aggregates. A complete binary tree with integer-labeled nodes suggests several aggregates: (a) the set of leaves, (b) the left subtree, and (c) a path from root to leaf. Each can be described as the set of nodes whose labels are related in some specific way.

In this section we introduce ***aggregate rewriting (AR) graph grammars***. An ***aggregate*** is a set of logically related nodes in a graph; for example (Figure 2), the leaves of a binary tree form an aggregate, as does its left subtree and a path from its root to a leaf. AR graph grammars rewrite entire aggregates in a single step (Figure 3). Each rewriting rule is extrapolated from a production that transforms a small, fixed size subgraph; thus arbitrarily large aggregates can be manipulated, allowing concise descriptions of recursively definable graph families.

Informally, a production of an aggregate rewriting graph grammar removes an aggregate from a host graph, transforms it, and reinserts the result by regenerating edges that provide the interface. Derivations in these grammars are similar to both those found in sequential and parallel graph rewriting systems: productions are parallel in their rewriting of nodes,

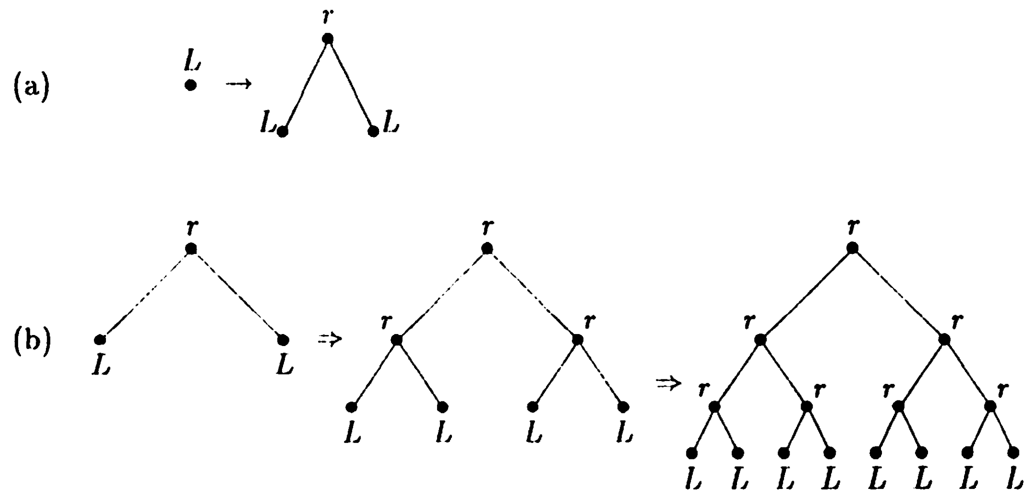


Figure 3: Aggregate manipulation. A production (a), which manipulates a single leaf of a binary tree describes the manipulation of all leaves. This production is used to grow successively larger, complete trees (b).

and sequential in their rewriting of aggregates. This dual nature is especially useful in the manipulation of large graphs. We now provide a more formal definition.

3.1 The Formalism

For the purposes of this paper, we work with undirected graphs without self-loops or multiple edges.[1] A *graph* is a system $G = (V_G, E_G, L_G, \gamma_G)$, in which V_G is a finite set of nodes, E_G is a set of two element sets on V_G, and L_G is a set of node labels identified with nodes by a total labeling function $\gamma_G : V_G \to L_G$. Graphs G and H are *isomorphic* if there is a bijection $\iota : V_G \to V_H$ which induces the natural bijection between E_G and E_H. An *occurrence of* G *in* H is a subgraph G' of H which is isomorphic to G; for the moment, we

[1] Although, the definitions may be subjected to obvious extensions.

assume that this isomorphism is label-preserving.[2]

An *aggregate* of graph G in graph H is a graph consisting of the union of the occurrences of G in H (Figures 4a–b). Since the aggregate consists of *all* occurrences of G in H, it is uniquely determined by graphs G and H. A ***aggregate rewriting production*** $P = (M, D, \phi, \pi)$ rewrites occurrences of a ***mother graph***, M, to copies of a ***daughter graph***, D, under the direction of an ***inheritance function*** $\phi : V_D \rightarrow V_M$. The inheritance function ϕ is a partial, surjective function that indicates, for some nodes of the daughter graph, a node in the mother graph that will provide interface edges. The function $\pi : \{1, ..., k\} \rightarrow 2^{\mathrm{dom}(\phi)}$, $k \geq 1$ defines a disjoint k-partitioning of the domain of ϕ, such that for each $1 \leq i \leq k$, the restricted inheritance function $\phi|_{\pi(i)}$ is surjective. A production is applied to a *host graph* yielding a ***image graph***.

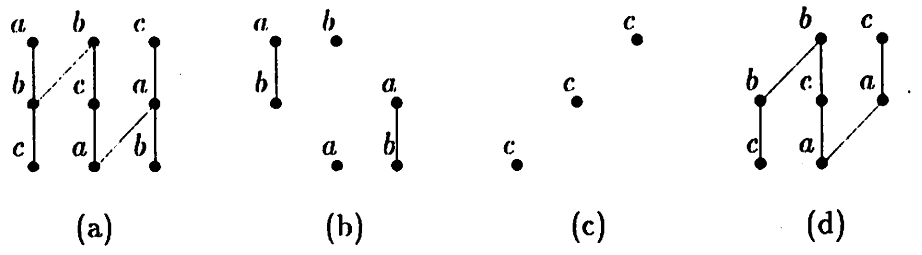


Figure 4: (a) A host graph with $a - b$ occurrences, (b) an aggregate of $a - b$ graphs, (c) the rest graph and (d) the interface. As is the case here, it is often useful to have graph occurrences overlap.

We now describe the mechanics of the parallel rewrite rule. All occurrences of the mother graph are removed from the host graph – yielding the ***rest graph*** (Figure 4). The ***interface*** is defined by those edges which either are incident to both the rest graph and an

[2]While it is common to require that the isomorphism between G and G' be label-preserving, it is not necessary. It is also possible to constrain the occurrence by other predicates as we discuss in a later section.

occurrence of the mother graph, or are incident to two distinct occurrences of the mother graph. For each occurrence of the mother graph found in the host graph, a daughter graph is disjointly added to the rest graph. The interface is rewritten using the following (Figure 5):

- If the edge $e = \{u, v\}$ is incident to the rest graph at u and an occurrence of the mother graph at v, an edge is introduced between u and all instances of nodes $v' \in \text{dom}(\phi) \subseteq V_D$ for which $\phi(v') = v$.

- If the edge $e = \{u, v\}$ is incident only to nodes in mother graph occurrences,[3] an edge is introduced between respective copies of the daughter graph, incident to instances of all pairs of nodes $u', v' \in V_D$ whenever $u', v' \in \pi(i)$ for some $1 \leq i \leq k$ and $\phi(u') = u$ and $\phi(v') = v$.

Various applications of the inheritance function are depicted in Figures 5b-d. In each example, the $a - b$ edges are inherited from interface edges between the rest graph and instances of the mother graph; the $b - b$ edges are inherited from interface edges between distinct instances of the mother graph. In (b), the inheritance function was trivially partitioned so all pairings of nodes in $\pi(1)$ from respective daughter graphs inherited the host $b - b$ edge; in (c), the inheritance function was nontrivially partitioned so only nodes in the same partition inherited the host $b - b$ edge; in (d), the inheritance function was not total and nodes labeled c did not inherit any edges.

An *aggregate rewriting graph grammar* (or *AR grammar*) is a system $G = (\Sigma, \Delta, P, S)$ where Σ is a finite, nonempty *label set*, $\Delta \subseteq \Sigma$ is a *terminal label set*, P is a finite set of aggregate rewriting productions, and S is a *start graph*. A graph H *directly derives* a graph K, written $H \Rightarrow K$, if there exists a production that transforms H into K as described previously. The reflexive, transitive closure of $\Rightarrow$ is written $\overset{*}{\Rightarrow}$, while the transitive closure

[3] Note that an interface edge may be twice incident to a single instance of a mother graph.

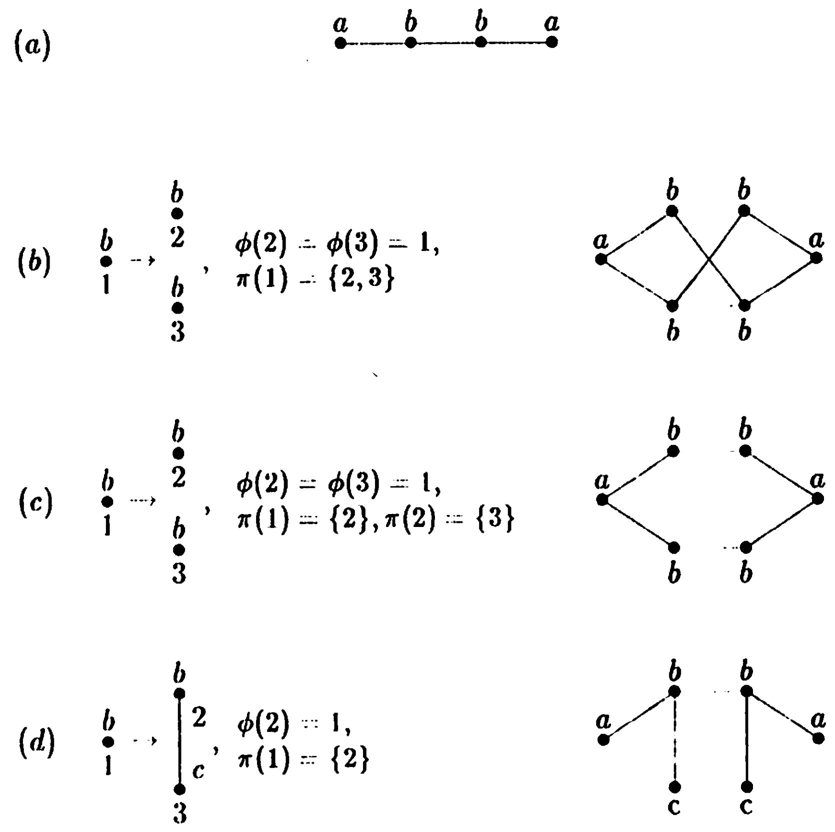


Figure 5: The various effects of inheritance functions on production application. The effect of rewriting the same host graph(a), using a total inheritance function without partitioning (b), a total inheritance function with partitioning (c), and a partial inheritance function (d).

is written $\overset{+}{\Rightarrow}$. A graph H *derives* K if $H \overset{*}{\Rightarrow} K$. H *derives* K *in n steps* if there exist K_i such that $H \Rightarrow K_1 \Rightarrow \cdots \Rightarrow K_n = K$. A graph K is a ***sentential form*** of a grammar $G = (\Sigma, \Delta, P, S)$ if $S \overset{*}{\Rightarrow} K$. The ***language of*** G is the set of all sentential forms that are labeled only from Δ.

3.2 Examples

In this section, we give specific grammars and discuss the general characteristics of AR grammars that they exemplify.

Example 1. *Binary Trees.* The grammar of Figure 6 generates the language of complete binary trees. It demonstrates the power of an individual production to manipulate arbitrarily large aggregates: productions are applied sequentially but each application operates in parallel on all leaves. Applications of the first production add a new level to the tree and applications of the second production terminate the derivation.

$$\Sigma = \{L, r, l\},\ \Delta = \{r, l\},\ S = \overset{L}{\bullet}$$

$$P = \left\{ \begin{array}{l} \left(\underset{1}{\overset{L}{\bullet}} , \underset{2}{\overset{L}{\bullet}} \text{---} \underset{3}{\overset{r}{\bullet}} \text{---} \underset{4}{\overset{L}{\bullet}} , \phi(3) = 1, \pi(1) = \{3\} \right) \\ \left(\underset{1}{\overset{L}{\bullet}} , \underset{2}{\overset{l}{\bullet}} , \phi(2) = 1, \pi(1) = \{2\} \right) \end{array} \right.$$

Figure 6: Grammar for the construction of complete binary trees. Figure 3 demonstrates a derivation using this grammar.

Example 2. *Binary n-cubes.* The grammar of Figure 7 generates the language of binary n-cubes. In the construction of an n-cube, two duplicates of the smaller $(n-1)$-cube are made and then grafted together by interface edges generated within a partition of the inheritance function; this process is shown in the derivation of Figure 8.

$$\Sigma = \Delta = \{C\},\ S = \overset{C}{\bullet}$$

$$P = \{(\underset{1}{\overset{C}{\bullet}},\ \underset{2}{\overset{C}{\bullet}} - \underset{3}{\overset{C}{\bullet}},\ \phi(2) = \phi(3) = 1, \pi(1) = \{2\}, \pi(2) = \{3\})\}$$

Figure 7: Grammar for the construction of n dimensional binary cubes.

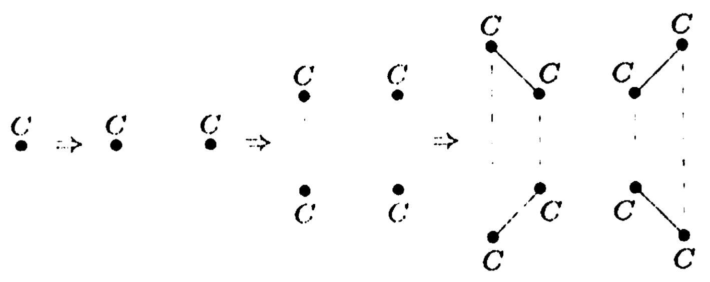


Figure 8: Derivation of various binary cubes. Note that solid edges are edges ***added*** by the production, dashed edges are ***inherited*** from previous cube instances.

Example 3. *Palindromes.* Some distinction can be made between various graph grammars by their ability to describe string languages[2]. The grammar of Figure 9 generates the language of all vectors labeled with palindromes over $\{a, b\}$. The first production generates a single character palindrome, the second production begins an even length string and the third production begins an odd length string. The fourth production extends vectors by adding a nonterminal to each of its ends and the remaining productions convert nonterminals to terminals. In each sentential form, any pair of nodes equidistant from the center must have identical labels and must, therefore, be rewritten at the same time by a single production. Note that a string grammar generating palindromes is quite different – it rewrites a single nonterminal in the center of the string.

$$\Sigma = \{A, B, C, a, b\},\ \Delta = \{a, b\},\ S = \overset{A}{\bullet}$$

$$P = \left\{ \begin{array}{l}
(\underset{1}{\overset{A}{\bullet}}, \underset{2}{\overset{C}{\bullet}}, \phi(2) = 1, \pi(1) = \{2\}) \\
(\underset{1}{\overset{A}{\bullet}}, \underset{2}{\overset{B}{\bullet}} \text{ — } \underset{3}{\overset{B}{\bullet}}, \phi(2) = \phi(3) = 1, \pi(1) = \{2,3\}) \\
(\underset{1}{\overset{A}{\bullet}}, \underset{2}{\overset{B}{\bullet}} \text{ — } \underset{3}{\overset{C}{\bullet}} \text{ — } \underset{4}{\overset{B}{\bullet}}, \phi(2) = \phi(4) = 1, \pi(1) = \{2,4\}) \\
(\underset{1}{\overset{B}{\bullet}}, \underset{2}{\overset{B}{\bullet}} \text{ — } \underset{3}{\overset{C}{\bullet}}, \phi(3) = 1, \pi(1) = \{3\}) \\
(\underset{1}{\overset{B}{\bullet}}, \underset{2}{\overset{C}{\bullet}}, \phi(2) = 1, \pi(1) = \{2\}) \\
(\underset{1}{\overset{C}{\bullet}}, \underset{2}{\overset{a}{\bullet}}, \phi(2) = 1, \pi(1) = \{2\}) \\
(\underset{1}{\overset{C}{\bullet}}, \underset{2}{\overset{b}{\bullet}}, \phi(2) = 1, \pi(1) = \{2\})
\end{array} \right.$$

Figure 9: Grammar for the construction of palindrome labeled vectors.

This grammar typifies the bias AR grammars have toward the generation of regular structures: it is easy to produce the set of palindromes but it is relatively difficult to produce the set of nonpalindromes. In fact, if we limit ourselves to single-node rewriting productions as in this example, non-palindromes cannot be produced at all. This is because a single non-terminal cannot rewrite nontrivially in the middle of the vector and preserve the ordered linear structure: if P is a production rewriting a single node to a connected, nontrivial daughter graph, P increases the degree of each inheriting node. Clearly, single node replacements must occur at the ends of the vector. If the nonterminals at the end of the strings are similarly labeled, the string will have the same prefix and (reversed) suffix; if they are labeled differently, then there is no control on the relationship between the left and right sides. The result is that a grammar that is either too conservative (generating only graphs which have the form *wvw*) or too liberal (generating graphs whose ends are independent – a superset of the nonpalindromes). Using more complicated AR grammars that are not limited to single node replacements, it is possible to describe the set of nonpalindromes.

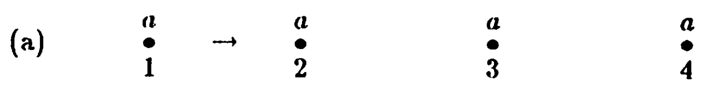
This grammar demonstrates an especially important property – the capacity for AR grammars to copy arbitrarily large structures. In the generation of a cube, the copies of aggregates were interconnected; it is also possible to create disjoint copies with appropriately partitioned inheritance functions, as in Figure 10.

Finally, we conclude this section with some general characteristics of AR grammars shared by all of the above grammars. Noting that inheritance functions are surjective, we have:

Observation 1 *AR graph grammars are node preserving: whenever $H \Rightarrow H'$, then $\#V_H \leq \#V_{H'}$.*

If a restriction of the inheritance function of a production P defines a graph isomorphism between a subgraph of the daughter graph and the mother graph, P is *monotonic*. For monotonic productions, we can extend this to edges as

Observation 2 *If every production of an AR graph grammar is monotonic, the grammar is edge preserving.*



$$\phi(2) = \phi(3) = \phi(4) = 1, \pi(1) = \{2\}, \pi(2) = \{3\}, \pi(3) = \{4\}$$

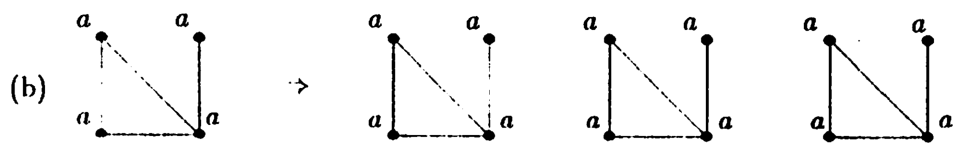


Figure 10: Partitioned inheritance functions aid in the copying of arbitrarily large aggregates. An aggregate, labeled everywhere '*a*', is copied three times by a production which partitions each inheritance to each of three disjoint nodes.

Grammars limited to single node productions *node aggregate rewriting graph grammars* (or *NAR grammars*) – form an important subclass of AR grammars and include all of the grammars in this section. Monotonicity is trivially met by NAR grammars and so we have

Observation 3 *NAR grammars are both node and edge preserving.*

NAR graph grammars are most closely allied with the constructions of node label controlled (NLC) grammars [3,4,5,6,7] and the parallel NLCp grammars. These grammar formalisms differ from NAR grammars, in that they admit some context – edges are preserved (or destroyed) based on labels of nodes neighboring the mother node. We have found that the uniform treatment of incident edges, while more restricted, leads to graphs with many of the characteristics described in Section 2.

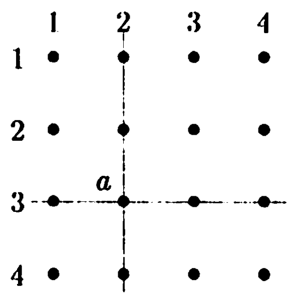


Figure 11: The node labeled 'a' has various identities: the ***node*** $[3,2]$, an element of ***row*** 3, and an element of ***column*** 2. Other labels that seem logical to the programmer define the set of labels that are semantically important in the transformations of the node.

4. Extensions to Aggregate Rewriting Graph Grammars

In this section we suggest some labeling extensions that make use of information one might expect to find in graph editors.

Labeling Using Sets

Graph editors perhaps provide richer descriptions of graphs than are considered in theoretical discussions. For example, a single node may have many labels which are suggested by viewing the structure in different ways. Figure 11 depicts a grid with a node labeled 'a'. As this node is logically a member of both row three and column two; manipulations of row three or column two should involve node a. As AR graph grammars partition nodes into disjoint aggregates (based on a single label), they are incapable of supporting the various views of the graph.

Thus, an extension to the formalism we have suggested is labeling of nodes from 2^{Σ}. Instead of label preservation, a natural constraint on the occurrence bijections is then label-membership, *i.e.*, if $\iota : G \rightarrow G'$, then $\gamma_G(v) \in \gamma_{G'}(\iota(v))$.

Predicate Labeling

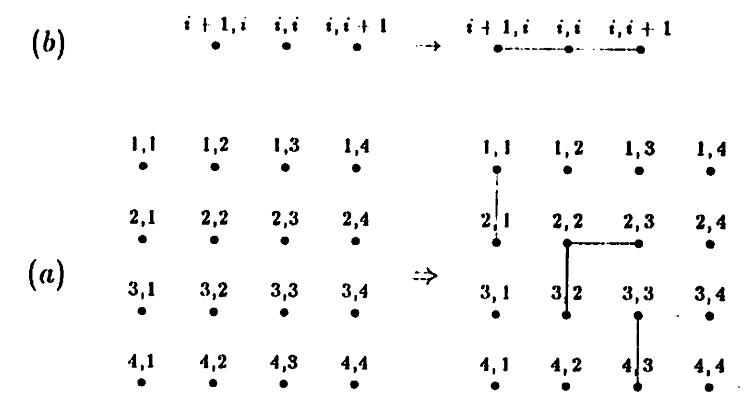


Figure 12: (a) A predicate labeled production which connects three nodes. (b) The effect of the production on a grid. Only three point subgraphs with consistent labeling are rewritten.

Aggregates might also be specified as a restriction on label values. For example, if we label nodes with integers, we might select even nodes by the label expression $i \cdot 0$ which requires the binary representation to end in zero. If i is considered a free variable, then both nodes of the left and right sides of a production are labeled using expressions involving i. When the free variables found on the mother graph of the production are satisfied by labels in the host graph, an occurrence is found, and the transformation inserts daughter graphs labeled in a manner that is consistent with the binding of the free variables. Figure 12 demonstrates the use of this labeling technique.

The Node Identification Filter

Rozenberg has suggested the use of 'filters' to restrict graph languages[9]. For example, labeling of graphs may obscure the underlying structure which may be important to analysis of the grammar. As communication structures are composed of distinguishable elements, identification of similarly labeled nodes in graphs proves useful. We extend the traditional notion of filter to a mechanism that filters sentential forms. Figure 13, depicts a use of an *identification filter*: in each sentential form one or more nodes of each daughter

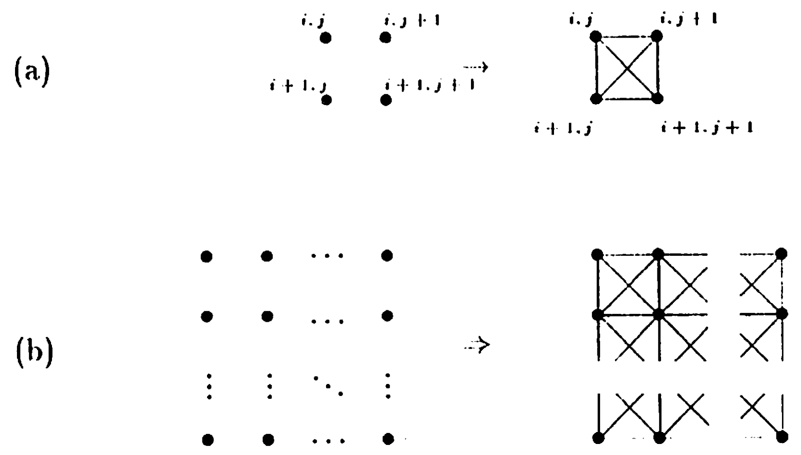


Figure 13: A production that transforms a totally disconnected set of points labeled (i, j) into an octagonal connected mesh. This uses free variables to identify relations between nodes, and a node identification filter to reunite similarly labeled disjoint nodes in the image graph.

graph are identified with daughter graphs whose nodes are similarly labeled. The need for identification occurs when the disjoint daughter graphs are re-introduced into the host graph. It is often useful to have two occurrences rewrite to overlapping graphs, or for a single occurrence to rewrite to a graph that forms new edges with the rest graph.

We are investigating the power of AR graph grammars that introduce the identification filter in production application.

5. Conclusion

The process of describing communication structures for highly parallel computation is in desperate need of automation. Communication graph editors are an important step toward making the specification of large communication structures viable. We have identified characteristics common to many statically programmed communication structures; these characteristics suggest biases a communication graph editor must have in constructing families of graphs.

Aggregate rewriting graph grammars represent a conservative parallel graph rewriting system that provides much of the support necessary for communication graph editors. In this paper we have introduced AR grammars and demonstrated some of the characteristics of grammars admitting node aggregate rewriting.

While such graph grammar formalisms can aid in the construction of efficient editors, their use of single node labels may be too restrictive to construct some common communication structures. We are now investigating the use of filters on these grammars, and their power in providing global information in localized rewriting systems such as aggregate rewriting graph grammars.

Acknowledgments. We would like to thank Grzegorz Rozenberg for helpful comments on this material.

References

[1] J. C. Browne, A. Tripathi, S. Fedak, A. Adiga, and R. Kapur. A language for specification and programming of reconfigurable parallel computation structures. In *1982 International Conference on Parallel Processing*, pages 142–149, August 1982.

[2] D. Janssens and G. Rozenberg. A characterization of context-free string languages by directed node-label controlled graph grammars. *Acta Informatica*, 16:63–85, 1981.

[3] D. Janssens and G. Rozenberg. Decision problems for node label controlled graph grammars. *Journal of Computer and Systems Sciences*, 22:144–177, 1981.

[4] D. Janssens and G. Rozenberg. Graph grammars with neighbourhood-controlled embedding. *Theoretical Computer Science*, 21:55–74, 1982.

[5] D. Janssens and G. Rozenberg. On the structure of node-label-controlled graph languages. *Information Sciences*, 20:191–216, 1980.

[6] D. Janssens and G. Rozenberg. Restrictions, extensions, and variations of NLC grammars. *Information Sciences*, 20:217–244, 1980.

[7] D. Janssens, G. Rozenberg, and R. Verraedt. On sequential and parallel node-rewriting graph grammars. *Computer Graphics and Image Processing*, 18:279–304, 1982.

[8] Hungwen Li, Ching-Chy Wang, and Mark Lavin. Structured process: a new language attribute for better interaction of parallel architecture and algorithm. In *1985 International Conference on Parallel Processing*, pages 247–254, August 1985.

[9] G. Rozenberg. Dependence graphs. In *Proceedings of the Third International Workshop on Graph Grammars (to appear)*, December 1986.

[10] Lawrence Snyder. Introduction to the configurable highly parallel computer. *Computer*, 15(1):47–56, January 1982.