An Approach to Programming Process Interconnection Structures: Aggregate Rewriting Graph Grammars*

Duane A. Bailey
Janice E. Cuny

COINS Technical Report 87-25
April 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

We describe a mechanism for generating families of process interconnection structures. Parallel programming environments that support individually programmed processor elements should allow the programmer to explicitly specify the necessary channels of communication at the level of logical abstraction of the algorithm. For highly parallel processors, the specification of this structure with traditional methods can be tedious and error-prone. *Aggregate rewriting graph grammars* provide a framework for describing families of regular graphs. Using this scheme, the difficulty of specifying an algorithm's communication structure is independent of its size. In addition, we note that *scripts* of derivation sequences generating different members of a family of structures can suggest an intra-family contracting map.

1. Introduction

When the processing elements of a parallel processor are individually programmed, explicit description of the necessary channels of communication often aids in the correct and efficient implementation of an algorithm. Knowledge of the underlying *communication structure* of an algorithm can provide redundancy needed for automatic error detection and correction, and it can provide structural information useful in mapping to a target architecture. Only a few parallel programming environments, however, support the explicit specification of communication structures[8,11].

Programmers are most effective when they work at the level of abstraction required by the algorithm. For programmers of parallel algorithms, this means communication structures should be logically depicted as graphs, as these representations usually accompany informal presentations of parallel algorithms. Graphical representations of communication structures serve as a basis for the display of mapping, control and debugging information. Programming environments that support graphical specification are currently very rudimentary: they rely on the programmer to draw each interconnection. This manual process is tedious, error prone and not feasible for large architectures. In addition, there is currently no support for the abstraction of *families* of communication structures. Description of graph families is necessary because most algorithms are designed in-the-small but are intended for arbitrarily large machines.

In this paper, we present a new form of graph grammar – called an aggregate rewriting graph grammar – and demonstrate its use in the specification of families of regular communication structures. This type of grammar facilitates description of regular structures at the programmer's level of abstraction. The resulting description is natural, compact and, in the case of recursively constructed graphs, a description that suggests contracting quotient maps[4].

In the next section, we informally describe aggregate rewriting graph grammars. The third section demonstrates the use of these grammars in describing a number of common

network structures. The fourth section suggests mapping techniques naturally induced by recursive descriptions. Our final section discusses the use of aggregate rewriting grammars within a programming environment for highly parallel computation.

2. Aggregate Rewriting Graph Grammars

The use of graph grammars in Computer Science has been largely restricted to describing transformations on structures that are easily represented by graphs: databases, derivation trees of a compiler, operations on abstract data types, *etc.* These systems do not, in general, have the regularity that we would expect to find in process interconnection structures. For our domain, we have been able to define a restricted form of graph grammar that introduces and preserves regularity and thus forms a natural basis for our descriptions.

An aggregate rewriting graph grammar (subsequently, an AR grammar) is a sequential graph rewriting mechanism[3,10]. The subgraphs to be rewritten at each production step are *aggregates* of nodes – the union of occurrences of a production's left side – allowing massive, but regular, changes in the structure of the transformed graph.

Terminology

The labels on nodes of our graphs consist of a major label and a subscript. The subscript is an n-tuple of strings.[1] We assume that nodes are uniquely labeled: a graph containing two nodes with the same label is equivalent to the graph built by identifying those nodes as in Figure 1. For the purposes of this work, we also assume graphs have no duplicate arcs, and that each arc is undirected.

Each production identifies a *mother graph*, which is rewritten to a *daughter graph* similar to that of NCE grammars[6]; when the mother graph is restricted to a single node (node

[1] The set of strings, along with valid operations on those strings (such as the operations of concatenation and addition we use in this paper), is determined by the designer of the grammar.

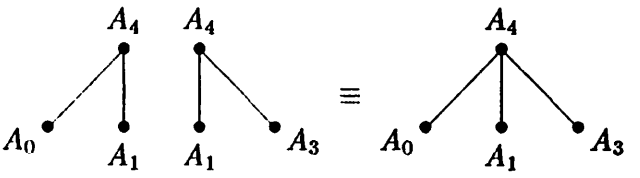


Figure 1: The two graphs shown above are equivalent; nodes generated with equivalent labels are identified.

aggregate rewriting graph grammars, or NAR grammars) these grammars are similar to NLC grammars[7]. The application of a production rewrites an *aggregate* – the set of all occurrences of the mother graph in the host graph.[2] The aggregate is removed from the host graph, a distinct daughter graph is created for each instance of the mother graph, and the union of these new graphs is re-embedded into the remainder of the host graph. Unlike string grammars, however, there are often a number of possible re-embeddings; the arcs connecting the aggregate to the remainder of the host graph and the arcs connecting instances of the mother graph – called *interface arcs* – must be inherited by the image graph in some consistent manner. This is described by a *inheritance function* (or *connection function*[7]) which identifies the mapping of interface arcs between occurrences of mother and daughter graphs.

The inheritance function for AR grammars is a partial surjective function, ϕ, from the nodes of the daughter graph to the nodes of the mother graph; it has been described more formally elsewhere[1]. Informally, if $\phi(u) = v$ then all edges incident to instances of the node v of the mother graph are inherited by respective instances of the node u in the daughter graph. When two or more nodes of a daughter graph inherit edges from the same node, the inheritance function may be *partitioned* (written $\phi = \sum_i \phi_i$) to indicate that images of the two nodes will never share inherited edges. Copies of edges incident to two mother graph occurrences may never cross the partitioning of the inheritance function.

[2]These occurrences may be constrained: for example, in this paper we assume a relation between the labels of a mother graph and its occurrence, and we extrapolate labels from the daughter graph that are consistent with each rewritten occurrence of the mother graph.

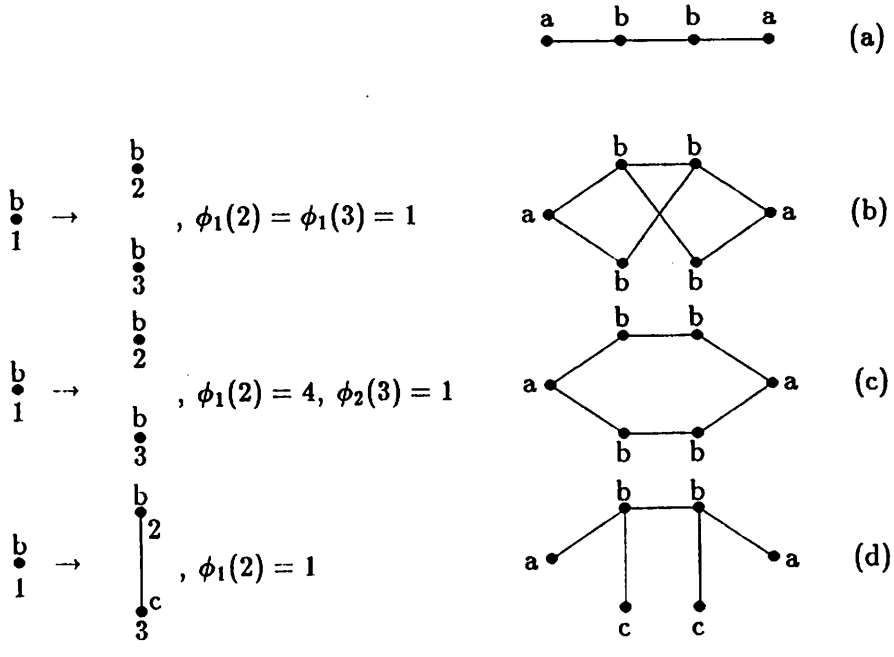


Figure 2: The various effects of inheritance functions on production application. The effect of rewriting the same host graph (a), using a total inheritance function without partitioning (b), a total inheritance function with partitioning (c), and a partial inheritance function (d).

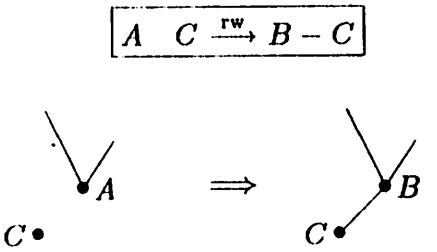
The effects of various inheritance functions are depicted in Figure 2. A host graph containing two occurrences of the node b is shown in (a). In (b), the inheritance function was not partitioned so all pairings of inheriting nodes from different daughter graphs inherited a copy of the host $b - b$ edge; in (c), the inheritance function was partitioned so only node instances from the same partition are incident to the same $b - b$ edge; and in (d), the inheritance function was not total, thus nodes labeled c did not inherit any edges.

A graph grammar generates graphs in the same way that a string grammar generates strings: a start-graph is iteratively rewritten by productions until each node of the graph is labeled with terminal symbols. The *language* of a grammar is the set of all terminally labeled graphs that can be generated from a start-graph.

Production types

For the remainder of this paper, we assume the productions of AR grammars not allowed arbitrary connection functions, but rather are restricted to three production types that differ in the inheritance of interface arcs. These have the following semantics.

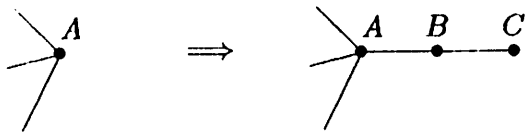
- **Relabeling productions.** The nodes of the mother graph are possibly relabeled and the injective mapping from mother to daughter nodes serves as the inheritance function. Thus, in the figure below, A is relabeled B and as a result, B inherits the arcs of A.



- **Extension productions.** The mother graph is rewritten to a larger daughter graph, as shown. An injective function from the mother graph to the daughter graph serves

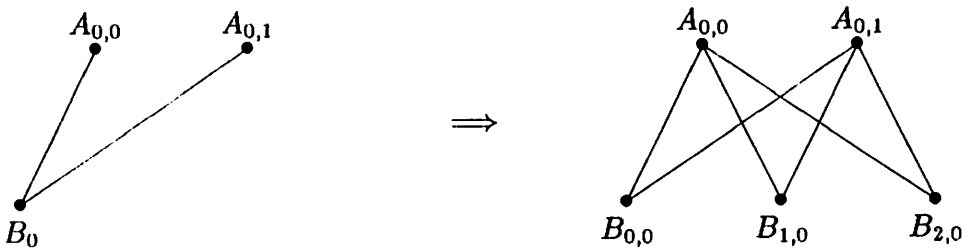
as the inheritance function. Excess nodes are not mentioned by the interface arcs and thus serve to 'extend' the host graph.

$$A \xrightarrow{\text{ext}} A - B - C$$



- **Replication production over S.** If S is a finite subset of the set of strings, the daughter graph consists of exactly $n = |S|$ copies of the mother graph; daughter graph nodes are labeled with the respective labels of the mother nodes, each prefixed with a distinct member of S. The power of this production type is detailed below: this replication production makes multiple copies of the node B_0 whose labels are prefixed with members of the string set $\{0, 1, 2\}$. Each receives a copy of the interface arcs mentioning B_0.

$$B_0 \xrightarrow{\text{rpl}\{0,1,2\}} B_{0,0} \quad B_{1,0} \quad B_{2,0}$$



Aggregates

The regularity of a communication network is often reflected by the labeling of its nodes. For example, the binary n-cube structure can be generated by labeling each of 2^n

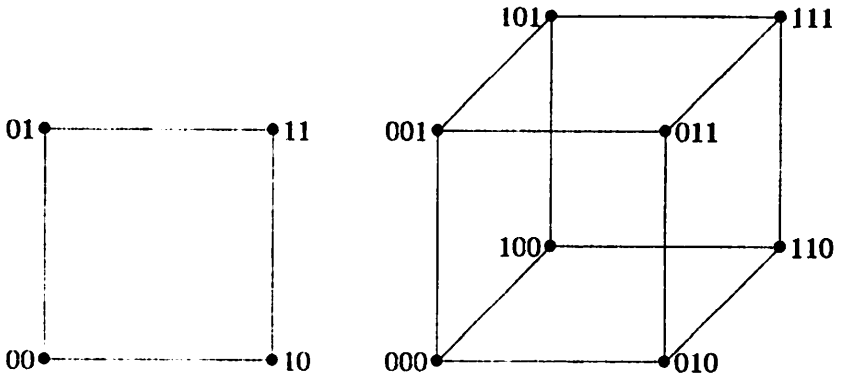


Figure 3: Two and three dimensional binary cubes. Each node is connected to all other nodes whose addresses differ in a bit. The 3-cube can easily be constructed by grafting two 2-cubes together.

nodes with a binary number and connecting nodes whose labels differ in exactly one bit (see Figure 3). Aggregate rewriting graph grammars make use of the regularity of subscripts in labels by allowing productions to rewrite *aggregates* of nodes that are similarly labeled. A label specification may contain *variables* which potentially match labels that induce an *assignment*. For example, if string concatenation is written $S \cdot T$, then the label specification

$$S \cdot 0$$

matches strings ending in 0. A set of label specifications identifies an aggregate of nodes if a set of the nodes suggest a consistent assignment of variables in the specification. Thus, the specification

$$A_{0 \cdot S} \text{ -- } A_{1 \cdot S}$$

matches all pairs of A-labeled nodes differing in exactly the first digit, which also share an arc. This specification can be used, for example, to identify the diagonal arcs of Figure 3.

Because mother and daughter graphs of productions specify aggregates, they become powerful rules for rewriting arbitrarily large structures simultaneously. Thus, while AR

grammar productions are applied sequentially, they rewrite many subgraphs of a host graph in parallel. The result is the union of the daughter graphs, appropriately embedded.

In the next section, we present a number of examples of regular communication structures which are generated by AR grammars.

3. Examples

In this section we demonstrate the power of AR grammars for describing families of regular communication structures. The set of strings for generating labels is the set of digits in an appropriate base (usually binary). We shall assume nonterminal labels are upper-case roman letters (*e.g.* 'T'), while a single terminal label Ω is used. To aid in the interpretation of these grammars, we supplement these examples with *scripts* which indicate derivation sequences which generate the desired family of structures.[3]

Binary Trees

Several methods of generating binary trees are possible – we demonstrate two. The first is a 'leaf-weighted' construction (Figure 4), which appends a new layer of leaves on a n-level complete binary tree to generate an $(n+1)$-level successor. The labeling of this tree is such that level n is labeled with n digit binary numbers. The children of a node are determined by appending either a 0 or 1 on the node's label. The script for the leaf-weighted grammar indicates that the family can be derived by the derivation sequences $(123)^n4$. Figure 5 shows a derivation for $n = 2$. The semantics of the production sequence $1 \cdot 2 \cdot 3$ is to generate a new layer of leaves, while production 4 terminates the derivation.

The second method (Figure 6) is 'root weighted': it generates a $(n+1)$-level binary tree by generating two copies of an n-level binary tree, and then constructing a common root (Figure 7). From the script we note that $1 \cdot 2 \cdot 3$ is the sequence of production

[3]Scripts *do not* indicate all possible production sequences, however, they *do* generate all graphs in the language.

Leaf-weighted Tree			
Start-graph: T_λ			
1	T_S	$\xrightarrow{\text{ext}}$	$T_S - X_S$
2	X_S	$\xrightarrow{\text{rpl}\{0,1\}}$	$X_{0,S}$ $X_{1,S}$
3	$X_{i,S}$	$\xrightarrow{\text{rw}}$	$T_{S \cdot i}$
4	T_S	$\xrightarrow{\text{rw}}$	Ω_S
Script: $(123)^n 4$			

Figure 4: 'Leaf-weighted' tree description. An n-level binary tree is constructed by adding a new level of leaves.

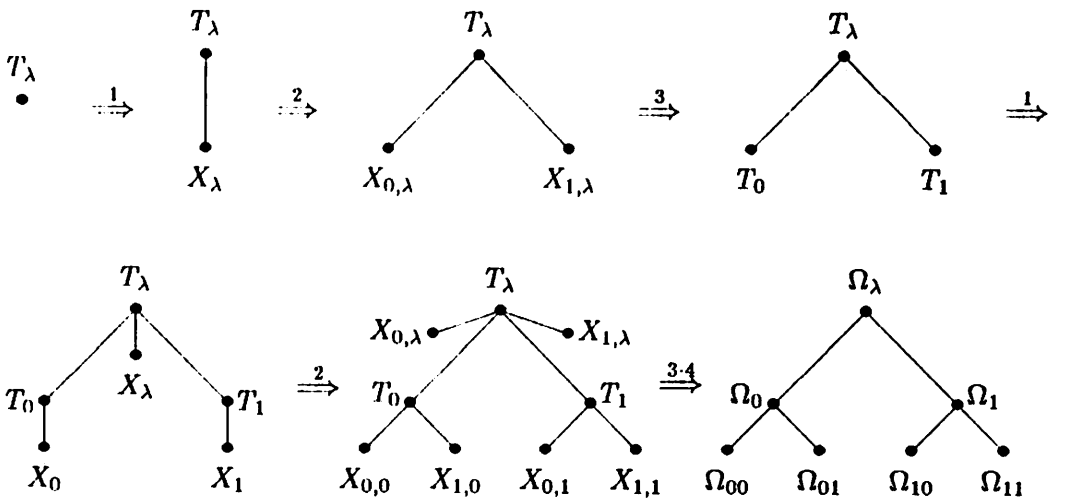


Figure 5: An example derivation of the grammar in Figure 4. Productions 1–3 add a new layer of leaves to the tree. Production 4 rewrites the labels to terminals. Note how $X_{0,\lambda}$ and $X_{1,\lambda}$ are identified with existing nodes T_0 and T_1 in the last steps.

Root-weighted Tree				
Start-graph: T_λ				
1	T_S	$\xrightarrow{\text{rpl}\{0,1\}}$	$T_{0,S}$	$T_{1,S}$
2	$T_{i,S}$	$\xrightarrow{\text{ext}}$	$T_{i,S} - X_S$	
3	X_S	$\xrightarrow{\text{rw}}$	T_S	
4	T_S	$\xrightarrow{\text{rw}}$	Ω_S	
Script: $(123)^n4$				

Figure 6: A 'root-weighted' tree grammar. A n-level binary tree is constructed by joining two copies of an $(n-1)$-level binary tree by mentioning a common root.

steps that increases the height by copying the tree (production 1) and generating a new root (production 2). While the labeling of this tree is identical to that of 'leaf-weighted' generation, we will see in the next section that the different generation of leaf- and root-weighted trees causes them to have distinct mapping characteristics.

Cubes

The binary (in general, m-ary) n-dimensional cube is constructed with the grammar depicted in Figure 8. The labels on the vertices of the n-dimensional binary cube structure are binary strings from 0 to $2^n - 1$. Each node is linked to vertices that differ in exactly one bit position. As the script shows, the productions 1 and 2 duplicate and connect smaller cubes to form larger cubes. Because each production matches an aggregate of nodes, the productions have the same semantics for arbitrarily sized host graphs.

Perfect shuffle

The perfect shuffle is described by the grammar found in Figure 9–a. This follows the original construction of Stone[12], which creates two copies of 2^n nodes (here, L_i

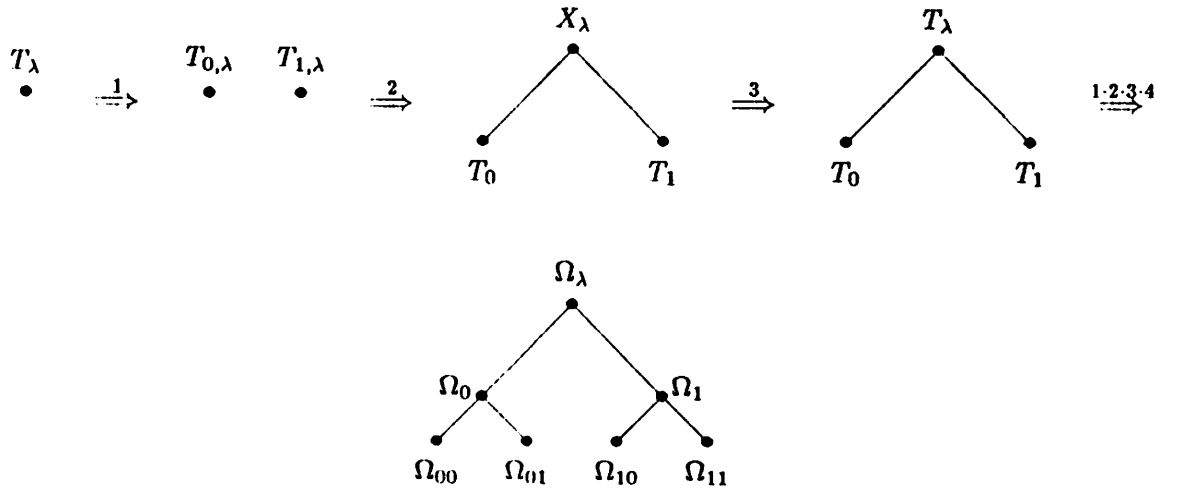


Figure 7: An example derivation of the 'root-weighted' tree grammar depicted in Figure 6. Productions 1–3 duplicate the tree and construct a common root.

m-Ary n-Cube				
Start-graph: C_λ				
1		C_S	$\xrightarrow{\text{rpl}\{0,\ldots,m-1\}}$	$C_{0,S} \cdots C_{m-1,S}$
2	$C_{i,S}$	$C_{(j \neq i),S}$	$\xrightarrow{\text{rw}}$	$C_{S \cdot i} - C_{S \cdot j}$
3		C_S	$\xrightarrow{\text{rw}}$	Ω_S
Script: $(12)^n 3$				

Figure 8: An m-ary n-cube grammar. The m copies of a m-ary, $(n-1)$-cube are joined together in productions 1 and 2.

and R_i) shuffles them, adds exchange arcs, and then reunites the relabeled copies. The script generates the 2^n copies of L and R nodes with n applications of productions 1 and 2. Production 3 generates shuffle arcs, while production 4 generates exchange arcs. Identification of similarly labeled nodes causes the L_i and R_i to be thought of as the same logical node. Figure 9-b shows the identification of logical nodes.

We have seen in this section that AR grammars are capable of describing a variety of regular structures. Other structures, not presented here, such as SW-banyans[5] and cube connected cycles[9] have also been described with AR grammars in a similar fashion. We have found that manipulation of labels is an extremely powerful capability. In the next section we will show that the structure of the script can provide an aid to the mapping of large logical communication structures into small processor arrays.

4. Mapping Techniques

One of the most important problems designers of parallel programming environment must address is the mapping of logical communication structures onto the physical communication network. It is often the case that the processor is too small to adequately handle the processes required by the algorithm.[4] Our description of *families* of communication structures does, however, allow us to identify special mappings that contract logical communication structures onto processors.

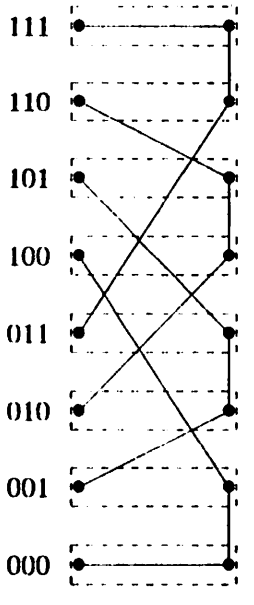
Fishburn and Finkel suggested a number of *quotient mappings* for common communication structures[4]. Essentially, if a n-process problem was to be mapped to a smaller m-processor array of the same family, a quotient map partitions the problem into m equivalence classes which are then mapped directly to the processor array.[5] Most of these map-

[4] Another problem occurs when the communication structure is not a member of the family of structures that may be easily mapped into the hardware. We will not discuss intra-family mappings induced by these descriptions in this paper.

[5] Ideally, this maps an identical number of processes to each processor, and maps the same number of

Perfect Shuffle							
Start-graph: L_λ R_λ							
1	L_S	R_S	$\xrightarrow{\text{rpl}\{0,1\}}$	$L_{0,S}$	$L_{1,S}$	$R_{0,S}$	$R_{1,S}$
2	$L_{i,S}$	$R_{i,S}$	$\xrightarrow{\text{rw}}$	$L_{i \cdot S}$	$R_{i \cdot S}$		
3	$L_{i \cdot S}$	$R_{S \cdot i}$	$\xrightarrow{\text{rw}}$	$\Omega_{i \cdot S}$ –	$R_{S \cdot i}$		
4	$R_{S \cdot 0}$	$R_{S \cdot 1}$	$\xrightarrow{\text{rw}}$	$\Omega_{S \cdot 0}$ –	$\Omega_{S \cdot 1}$		
Script: $(12)^n 34$							

(a)



(b)

Figure 9: (a) Perfect shuffle grammar, and (b) node identification.

pings are identified by labeling the nodes of the graphs appropriately. For example, a square grid labeling each node with binary row and column binary indices from 0 to $2^n - 1$ can be reduced in size by dropping the least significant bit from both indices of each node.

Because AR grammars naturally suggest recursive descriptions of graph families the scripts provide a natural ordering of derivations. For example, a binary n-cube is generated by the derivation sequence $(1 \cdot 2)^n \cdot 3$. In the process of generating a $(n+1)$-cube, a smaller n-cube is first generated. Clearly, every node labeled S of an $(n+1)$-cube is attributable to exactly one node of an n-cube: the node labeled with n digit prefix of S. A natural mapping of a boolean $(n+1)$-cube into an n-cube is the one induced by this quotient map.

In general, recursively defined families have quotient maps suggested by their parameterized derivation sequences as described above. The implementor of the complete binary tree, for example, has the option of selecting either the leaf-weighted or root-weighted grammars which suggest the leaf-weighted and root-weighted quotient mappings defined by Fishburn and Finkel[4]. The ability to make this decision aids the user in fashioning a mapping of an arbitrarily large algorithm into the processor in a manner that is most efficient.

5. Conclusions

We have informally described AR grammars, which can be used to specify families of regularly structured graphs. In light of programming highly parallel processors, these grammars show promise in accurately describing a wide variety of structures. The power of aggregate rewriting productions allows the user to apply straightforward logical transformations on a structure in a manner that scales up. In addition, where computing resources are limited, the derivation scripts suggested by these grammars provide a useful basis for

logical channels to each physical channel. A counter-example can be found in complete binary trees, which fail to map evenly since $2^m - 1$ does not divide $2^n - 1$ for $1 < m < n$. However, augmenting trees with an extra node fixes this problem[2].

mapping large logical process structures onto smaller physical processor arrays.

As we expect highly parallel processors will soon be impossible to program by hand, graph construction tools must provide descriptive primitives that manipulate aggregates of nodes similarly. Furthermore, as the size of the problem increases we believe the user will find appropriate manipulation of labels a useful mechanism for generating node specific data. We believe aggregate rewriting graph grammars can provide support for future graphical interfaces to parallel programming environments.

References

[1] Duane A. Bailey and Janice E. Cuny. *Graph Grammar Based Specification of Interconnection Structures.* Technical Report 87-23, University of Massachusetts at Amherst, March 1987.

[2] Duane A. Bailey and Janice E. Cuny. *The Use of Shape Grammars in Processor Embeddings.* Technical Report A-86-23, University of Massachusetts at Amherst, July 1986.

[3] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *14th Conference on Switching and Automata Theory*, pages 167–179, 1973.

[4] John P. Fishburn and Raphael A. Finkel. Quotient networks. *IEEE Transactions on Computers*, C-31(4):288–295, April 1982.

[5] Rodney L. Goke. *Banyan Networks for Partitioning Multiprocessor Systems.* PhD thesis, University of Florida, 1976.

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

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

[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] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: a versatile network for parallel computation. *Communications of the ACM*, 300–309, May 1981.

[10] H. J. Schneider. *Graph Grammars*, pages 314–331. *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, September 1977.

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

[12] Harold S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, February 1971.