ParaGraph: Graph Editor Support for Parallel Programming Environments

Duane A. Bailey
Janice E. Cuny
Craig P. Loomis

COINS Technical Report 89–53
August 1989

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

ParaGraph: Graph Editor Support for Parallel Programming Environments [1]

Duane A. Bailey
Department of Computer Science
Williams College, Williamstown MA
bailey@cs.williams.edu

Janice E. Cuny
Craig P. Loomis
Department of Computer and Information Science
University of Massachusetts, Amherst MA
cuny@cs.umass.edu
loomis@cs.umass.edu

Abstract: We report here on a graph editor, called ParaGraph, that supports the programming of massively parallel computations. It provides a flexible mechanism for the concise specification of families of annotated graphs, addressing the problems of user-annotation and scale independent graph manipulation. It has been integrated into our programming environment, where it serves as the basis for tools supporting communication abstractions in program specification and debugging. Its extension to a number of other parallel programming environments would be straightforward.

Keywords: Editors, parallel programming environments, user interfaces, communication structures, massive parallelism, graph grammars.

1. Introduction

Programmers will require sophisticated support tools in order to develop massively parallel computations. Here, we focus on computations designed for message-passing architectures, considering support for fine grain parallelism in which large numbers of processes communicate frequently across regular interconnections. For these computations, it is apparent that programming tools should facilitate the use of graphs.

Graphs naturally represent visualizations used by algorithm designers. Pictures of graphs, for example, often accompany algorithm specifications in the literature. They

[1] The Parallel Programming Environments Project at the University of Massachusetts is supported by the Office of Naval Research under contract N000014-84-K-0647 and by the National Science Foundation under grants DCR-8500332 and CCR-8712410.

can be used to depict *process structures* showing potential channels of communication, or *global communication patterns* giving high-level views of interprocess communication behavior [3]. Providing support for the explicit specification of graphical representations can reduce the disparity between a programmer's conceptualization of his algorithm and its implementation. In addition, it can increase the homogeneity of process code [2], eliminate the possibility of some types of communication errors [3], provide a basis for coherent graphical displays [15,23], and facilitate the manipulations needed in scaling, animation, and debugging of parallel programs.

The use of explicit graph specifications in parallel programming environments is not new [6,7,8,16,17,20,21,22], but none of the existing tools provide the comprehensive support needed. Such support must include

* **Scalable graph specifications.** Programmers typically implement and debug small versions of their algorithms which are then scaled for massive parallelism. Even after the program is completed, it is often necessary to rescale it — both up and down — to reflect problem size or hardware availability. *Support for explicit graph representations should facilitate the description of not just graph instances, but entire graph families.*

* **User-specific annotations.** Interpretation of a graph requires the labeling of nodes and edges with user- or tool-specific attributes. Such attributes might, for example, define process code, actual parameter values, graphical rendition, resource assignment, channel names and protections, or communication protocols. *Support for explicit graph representations should provide concise, flexible mechanisms for the annotation of graphs.*

* **Graph compositions.** Often more than one graphical representation is used in a computation: a program may have phases of execution that require distinct graphs or it may be most naturally expressed as the projection of a number of partial graphs. As programmers become accustomed to massively parallel computing, composition will be used to form more complex programs. *Support for explicit graph representations should include mechanisms for graph composition.*

* **Graph visualizations.** In using graphs to represent the visualizations of algorithm designers, it is crucial that graph presentations be comprehensible. *Support for explicit graph representations should provide facilities for the management of graph layouts.*

Most of the existing tools allow the user to annotate graph nodes and draw their interconnections [17,20,21,22]. Few, however, provide any mechanisms for scaling or layout

assistance. Polylith [21] and CODE [7] provide some scalability, Polylith by instantiating components before interconnection and CODE by hierarchical descriptions that include replication nodes. Other approaches provide scalability either with textual descriptions [16], losing the benefits of graphical rendition, or with library selections [6], losing the flexibility needed for user-annotation. The EDGE graph editor [19], not designed specifically for parallel programming applications, does provide extensive layout assistance.

We report here on ParaGraph,[2] a graph editor that is unique in providing the comprehensive range of facilities needed to support the explicit use of graph structures in a parallel programming environment. Figure 1 shows a ParaGraph session in which the family of complete binary trees is specified (in the windows labeled `Start` and `TreeProd`) and two members of that family are generated (the windows labeled BINTREE(1) and BINTREE(2)). Details of this figure are discussed more fully later in the paper.

The implementation of ParaGraph is based on a graph grammar formalism specifically suited to the description of parallel communication structures. In Section 2, we describe that formalism. In Section 3, we illustrate ParaGraph's interface to basic graph grammar mechanisms and, in Section 4, we describe its extensions that facilitate programming. In Section 5, we discuss the role of ParaGraph within our parallel programming environment and, in Section 6, we summarize our contributions.

2. Theoretical Foundations

Our editor is based on a graph rewriting formalism called Aggregate Rewriting (AR) Graph Grammars [1,4]. We describe these grammars briefly, first giving an informal introduction and then giving the formal details.

2.1 An Introduction to AR Graph Grammars

An *aggregate* is a set of logically related nodes in a graph; for example, the leaves of a binary tree form an aggregate, as do its left subtree and a path from its root to a leaf:

---

[2]Though the prefix "para" might suggest parallel (either because we use a parallel graph rewriting mechanism or because we apply our results to parallel programming), we interpret it to mean "*beyond*" (as in "paranormal"), emphasizing the fact that the editor supports the specification of not just single graphs, but entire graph families.

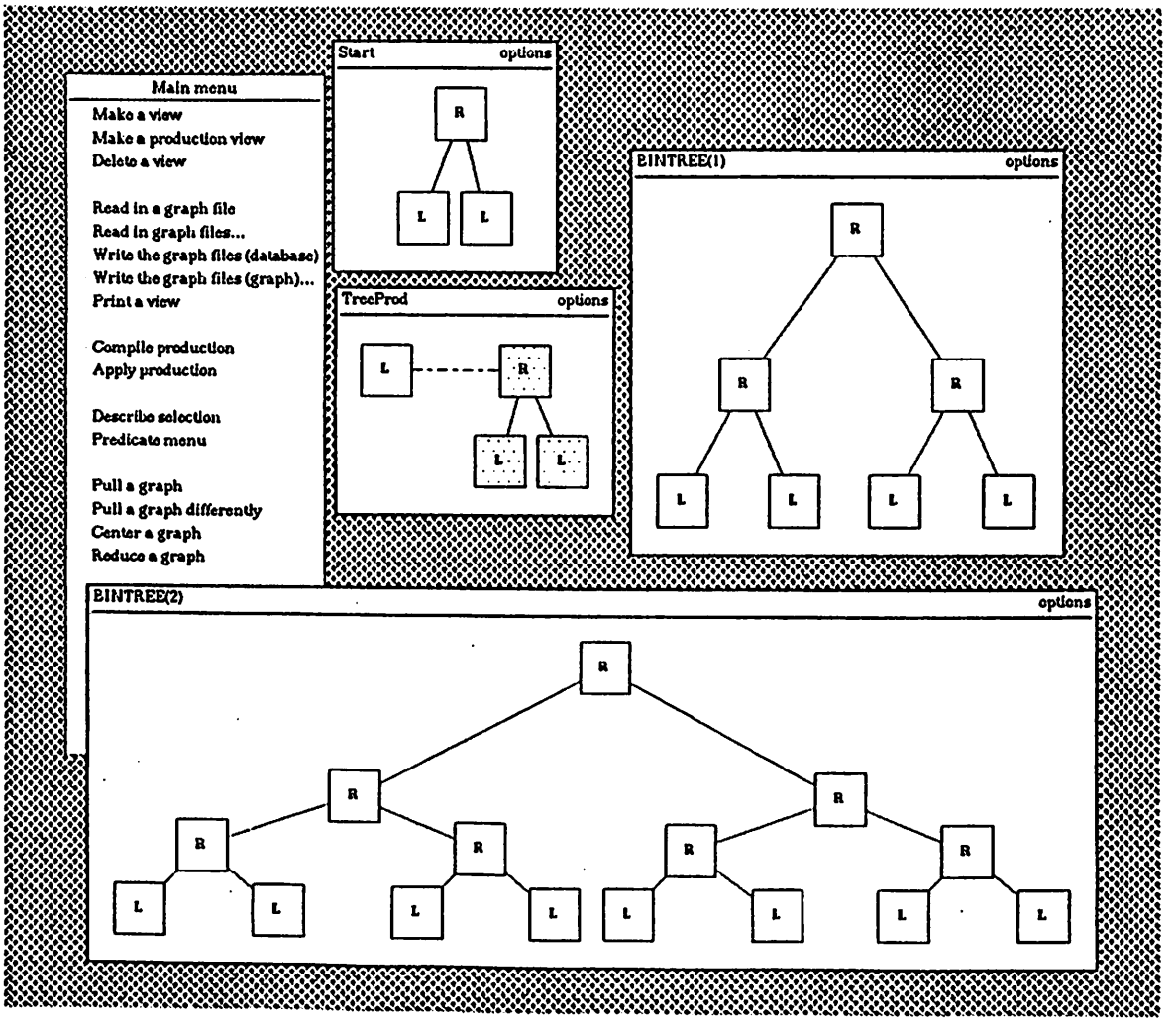
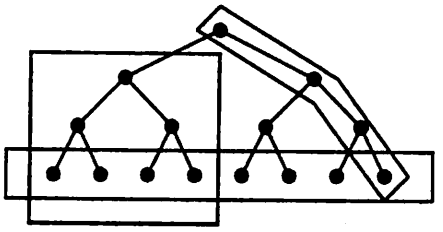
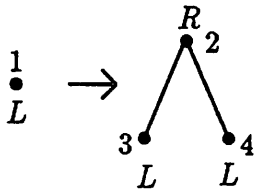


Figure 1: ParaGraph specification of the family of complete binary trees and the first two generated members of that family.

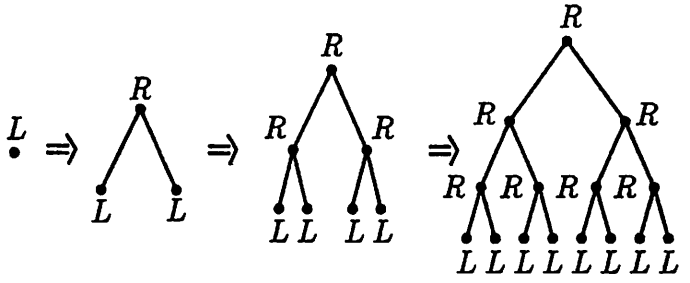


AR graph grammars rewrite entire aggregates in a single step. Each rewriting rule is extrapolated from a production that transforms a small, fixed size subgraph; in applying that production, the aggregate of all instances of its left-hand side is replaced by an aggregate of instances of its right-hand side. Each application of the production



$\phi(2) = 1$

for example, rewrites the entire frontier of an appropriately labeled tree:

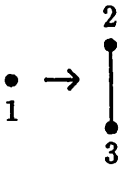


The production consists of two graphs. Using standard terminology, we call its left-hand side the *mother graph* and its right-hand side the *daughter graph*; the production is used to rewrite a *host graph* into an *image graph*. Nodes in the production are uniquely numbered for identification. The labels L and R used here are annotations specific to this grammar. In general, labels on the mother graph are interpreted as expressions containing free variables. Occurrences of the mother graph are isomorphic subgraphs whose labels match with consistent bindings of the free variables. In this case, the expression is a constant matching all nodes labeled L.

The function ϕ describes *inheritance* that will govern the introduction of edges connecting newly created daughter graph instances to each other and to the graph that remains after removal of the mother graph aggregate. For this production, ϕ indicates that an

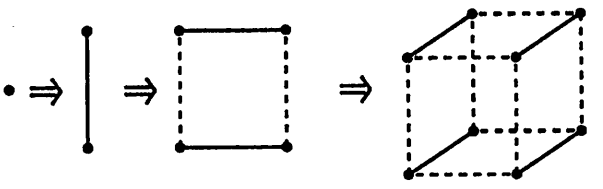
instance of the node labeled 2 (the root of a daughter graph) inherits edges from the corresponding instance of the node labeled 1 (the single node of a mother graph). In the second step of this derivation, for example, inherited edges connect the root with the first level of interior nodes.

In tree derivations from this grammar, the host graph never contains edges between instances of the mother graph (the aggregate of the leaves of a tree is a disconnected collection of nodes). When the host graph does contain edges between instances of the mother graph, which pairs of inheriting nodes in the corresponding daughter graphs should be connected? If all pairs are connected, the daughter graph instances are joined by crossbars; if no pairs are connected, the daughter graph instances are disconnected. For the structures normally encountered in parallel computation, neither alternative is appropriate: crossbars are expensive and unconnected nodes cannot coordinate their computations. In AR graph grammars, we introduced the notion of *partitioned inheritance* to provide a balance: the inheritance function is partitioned and inheritance respects partitioning (that is, pairs of inheriting nodes from corresponding daughter graphs are connected only when they inherit under the same partition).[3] Partitioning is used in the following production to generate the family of hypercubes:



$$\phi_1(2) = 1$$
$$\phi_2(3) = 1$$

The inheritance function here is partitioned into ϕ_1 and ϕ_2; instances of pairs of nodes corresponding to node 2 and node 3 are in the domain of different partitions and, therefore, will not be connected by inheritance. Using solid lines for edges introduced by the daughter graph and dashed lines for edges introduced by inheritance, this production results in the following derivation:

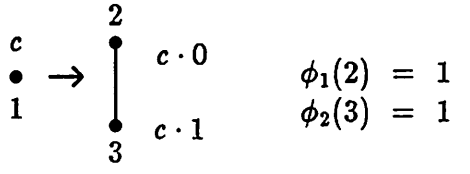


Partitioning allows the copying of graph structures. Each application of the cube production copies the existing graph and creates edges between corresponding nodes of the copies.

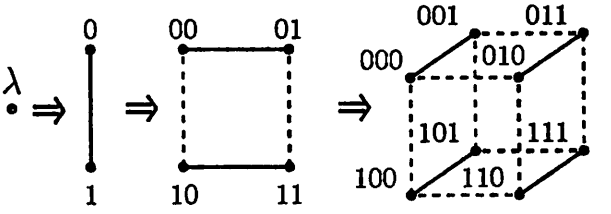
[3]This decision does not preclude the construction of either crossbars or disconnected daughter graphs.

Edges introduced by the daughter graph connect the copies; edges introduced by inheritance preserve the connections within each copy. Without partitioning, this production would have generated the family of complete graphs on 2^k nodes, for $k \geq 0$.

Nodes can be annotated as they are generated. Bindings of variables established in matching occurrences of mother graphs are used as context in labeling the corresponding instances of daughter graphs. Thus, for example, to generate "standard" binary addresses for the cube, the above (partitioned) production can be modified to



For each application of the production, the variable c is bound to the current label of the mother graph node and then used in generating the label of the replacing daughter graph nodes:



This brief introduction to AR grammars should be sufficient for the casual reader who may wish to skip the formal details presented next.

2.2 AR Graph Grammars: Formal Description

A *graph*[4] is a system $G = (V_G, E_G, L_G, \gamma_G)$, in which V_G is a finite set of vertices, E_G is a set of two element sets on V_G, and L_G is a set of vertex labels identified with vertices by a total labeling function $\gamma_G : V_G \rightarrow L_G$. Graphs G and H are *isomorphic* if there is a bijection $\iota : V_G \rightarrow V_H$ 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 assume that this isomorphism is label-preserving.

An *aggregate* $[G \subseteq H]$ is a graph consisting of the union of the occurrences of G in H (Figures 2a,b). A *aggregate rewriting production* $P = (M, D, \phi = \sum_i \phi_i)$ rewrites

[4]For the purposes of this paper, we work with undirected, connected graphs without self-loops or multiple arcs. Although, the definitions may be subjected to obvious extensions.

occurrences of a *mother graph*, M, to copies of a *daughter graph*, D, under the direction of an *inheritance function* $\phi : D \rightarrow M$. The inheritance function ϕ is a partial, surjective function that indicates, for some nodes of a daughter graph, a node in the mother graph that will provide connecting edges. It is often useful to consider a fixed partitioning of ϕ into one or more ϕ_i, each of which is also surjective.[5]

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 2). The

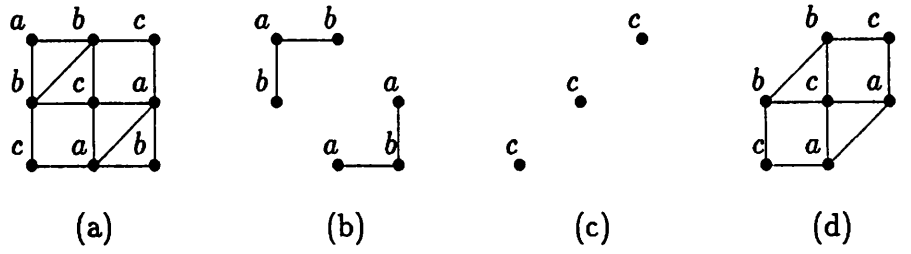


Figure 2: (a) A host graph with $a - b$ occurrences, (b) an aggregate of $a - b$ graphs, (c) the rest graph and (d) the interface. Note that graph occurrences overlap.

interface is the graph induced by the set of edges which either are incident to both the rest graph and an 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:

- 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 $v' \in V_D$ for which $v' \in dom(\phi_i)$ and $\phi_i(v') = v$.
- If the edge $e = \{u, v\}$ is incident to two occurrences of a mother graph, an edge is introduced between copies of the daughter graph incident instances of $u' \in V_D$ and $v' \in V_D$ whenever $u', v' \in dom(\phi_i)$ and $\phi_i(u') = u$ and $\phi_i(v') = v$.

Various applications of the inheritance function are depicted in Figure 3. In each example,

[5]The trivial partitioning $\phi_1 = \phi$ is used when no logical partitioning of ϕ is desired.

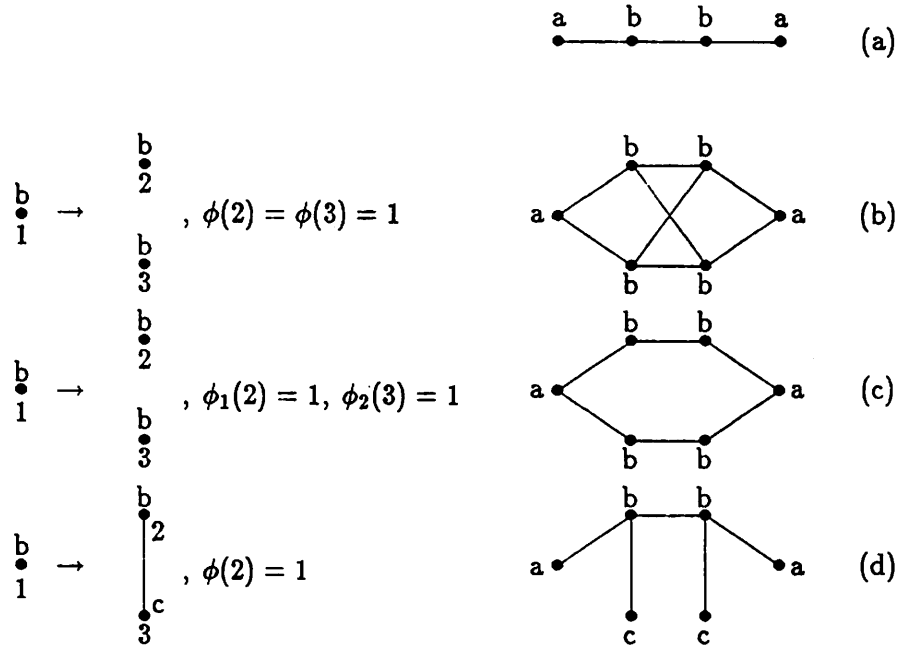


Figure 3: The various effects of inheritance functions on production application. The effect of rewriting the same host graph(a), using a total inheritance function without partitioning (b), a total inheritance function with partitioning (c), and a partial inheritance function (d).

the $a - b$ 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 not partitioned so all pairings of nodes in the domain of ϕ_1 from different daughter graphs inherited the host $b - b$ edge; in (c), the inheritance function was 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* is a system $G = (\Sigma, \Delta, P, S)$ where Σ is a nonempty *label set*, $\Delta \subseteq \Sigma$ is a *terminal label set*, P is a 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 $\stackrel{*}{\Rightarrow}$. A graph H *derives* K if $H \stackrel{*}{\Rightarrow} K$. A graph K is a *sentential form* of a grammar $G = (\Sigma, \Delta, P, S)$ if $S \stackrel{*}{\Rightarrow} K$. The *language of* G is the set of all sentential forms that are labeled only from Δ. (In our application, we ignore the set of terminal symbols and, instead, use patterns of production sequences to select the subset of sentential forms that are to be included in the language.)

In describing the formalism, we have assumed that nodes are labeled with single labels and that the isomorphism used in matching occurrences of the mother graph depends on simple label matching. Our applications demand multiple node labels and more complex rules for label matching and generation, requiring a logical extension of our formalism. For the purposes of this section, we have avoided this distraction.

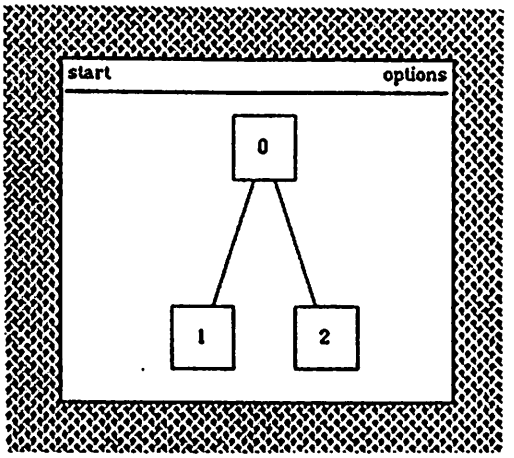
3. The ParaGraph Interface to AR Grammars

AR graph grammars provide a rewriting mechanism particularly suited to the construction of graphs commonly used in parallel programming (graphs typified by low degree, low diameter, and near symmetry). This mechanism is, however, difficult for programmers to use directly. The ParaGraph editor provides an accessible user interface.

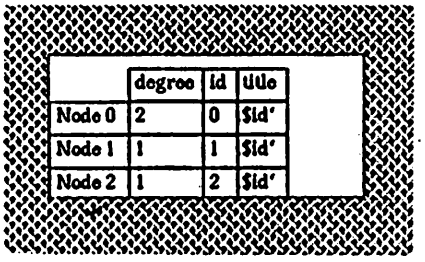
Using ParaGraph, the programmer begins by specifying the smallest member of his graph family. He then describes the set of transformations needed to convert that graph into the next larger family member and he develops a *script* to direct the order of their application. The initial graph becomes the start graph of an underlying graph grammar, the transformations become its productions, and the script determines allowable derivation sequences.

In this section, we illustrate ParaGraph's support for the basic AR rewriting mechanisms with two examples, corresponding to the complete binary tree and cube grammars given above.

Example 1: Complete Binary Trees. We begin by specifying the smallest, nontrivial member of the family of complete binary trees — a three node tree.[6] Individual graphs are drawn (using a mouse) in separate windows, called *views*:



Graphs can be annotated with both system- and user-defined attributes. The numeric labels here, for example, give the values of a system-defined attribute, `id`, which uniquely identifies nodes within a view. Since only limited amounts of information can be displayed on the graph itself, the values of all attributes are given in pop-up windows called *scorecards*. Initially, the scorecard for this graph contains



	degree	id	title
Node 0	2	0	$id'
Node 1	1	1	$id'
Node 2	1	2	$id'

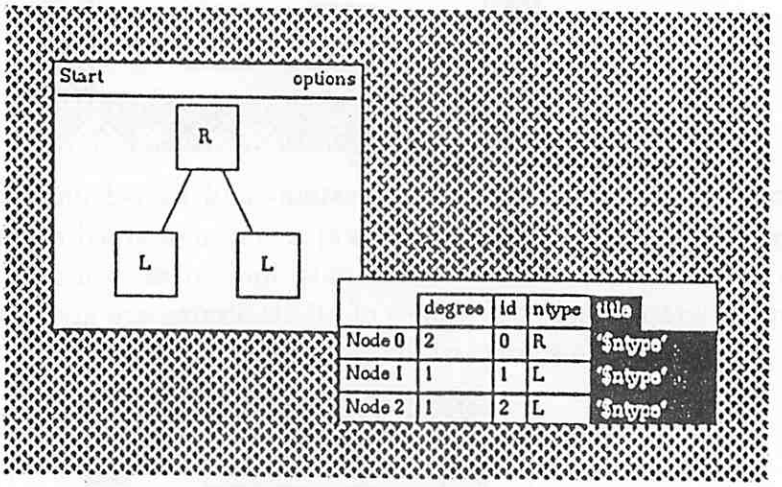
where `degree` has the usual graph theoretic meaning and `title` names the attribute, if any, to be displayed on the graph. User-defined attributes can be added by filling in a "global" attribute template. An attribute of string type, for example, can be created with

[6] If it were useful, we could just as easily have based our specification on the trivial, single node tree.

Create attribute (string value).
Create attribute (integral value).

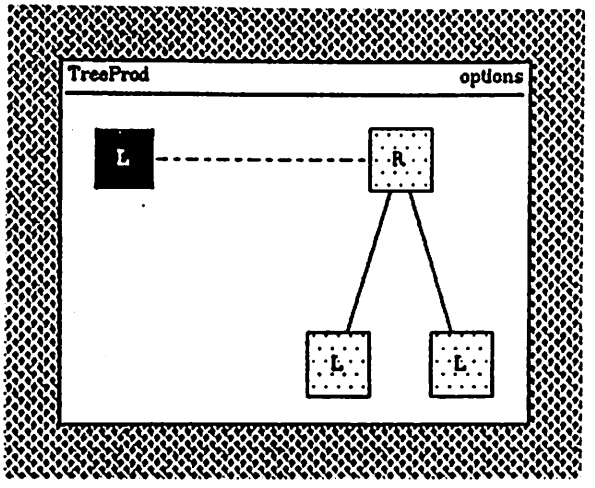
Context name :	Node
Attribute name :	ntype
Attribute type :	string
Default value :	

Once an attribute is introduced, it appears on the scorecard and can be given values (by editing the appropriate column):



For the binary tree specification, this labeling of the three-node graph completes the start graph. Next, we must describe the transformation needed to convert it into the seven node tree.

Eventually, we expect to provide a set of conceptually primitive transformations that will make the editor accessible to programmers not familiar with AR grammars. Currently, however, transformations are specified as productions. Each production is drawn in a special window, called a *production view*. Within that window, color is used to distinguish mother and daughter graph nodes (rather than the normal left/right orientation). For the purposes of this paper, we use dark, solid nodes for the mother graph and lighter nodes for the daughter graph. Annotation is done with scorecards. Inheritance is shown as a dashed line between a daughter graph node and the mother graph node from which it inherits:



This transformation corresponds to the single production of the tree grammar defined above. We can apply it to the start graph to generate the seven node graph which will be displayed in a new view. Application of the transformation is independent of host graph size, and, thus, we may iteratively reapply it to create any desired member of the tree family.

Derivation sequences for this grammar are simple: there is only one production and each of its applications yields a new family member. Often, however, grammars have multiple productions and not all derivation sequences produce meaningful graphs. Within the grammatical formalism, undesirable graphs are precluded with the use of terminal symbols and label matching, but often this is cumbersome. ParaGraph lets the user control graph generation directly with parameterized expressions, called *scripts*. Each script contains a start graph and an expression denoting allowable derivation sequences. Expressions range over the set of transformation names (taken from the title bars of production views) and use operations of concatenation and iteration. Concatenation is implicit. Iteration is indicated by parenthesis; each parenthesized expression is followed by a parameter that will determine the number of iterations used in a given derivation sequence. There can be more than one parameter in a script and a single parameter can be repeated.

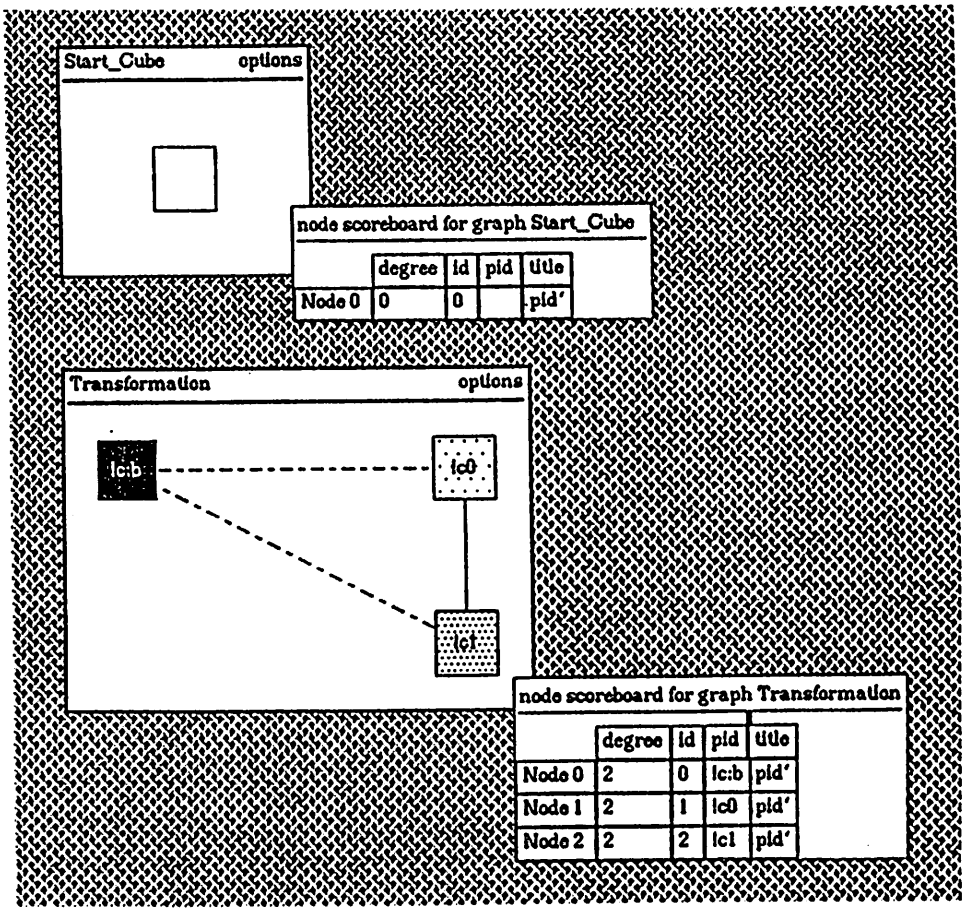
For the binary tree specification, the (trivial) script is

`Start : (TreeProd)n`

where `Start` and `TreeProd` are as defined above and the value of n determines the number of applications needed to generate a specific family member. Naming this script BINTREE, we can refer to the first two generated graphs, shown in Figure 1, as BINTREE(1) and BINTREE(2).

It should be noted that our transformation produces graphs which have the idiosynchratic "leaves-down" orientation. This stems from the "local" placement strategy ParaGraph employed during rewriting. It is similar to that used in shape grammar derivations [25] and takes in to account the orientation of the mother and daughter graphs as well as that of the matched mother graph instance. In this generation of the binary tree, we used an optional constraint which scales the "real estate" of the daughter graph instance to fit within the real estate of the mother graph occurrence. We expect that many other graphs will have recursive layouts paralleling their construction in this manner. Where such local node placement techniques are not successful, more traditional graph layout mechanisms can be invoked (as in the next example).

Example 2: Hypercubes. The ParaGraph specification that corresponds to the cube grammar given above has a trivial start graph and a single transformation:



Partitioning of the inheritance function is indicated by the shadings of the daughter graph nodes: all nodes with the same shading are in the domain of the same partition. (Partitions

are created by "selecting" their constituent nodes.)

For label matching and generation, ParaGraph uses a simple string matching mechanism [24]. As the attributes on mother graph nodes are matched, subexpressions are bound to variables providing the context for label generation; bindings are constrained to be consistent across a mother graph instance. In this example, the `pid` attribute is used for standard labeling of the cube. The expression

```
!c : b
```

is translated into an expression that matches any binary string. During the matching of the mother graph, the matched string is bound to the identifier `c` (`b` indicates Boolean). During the generation of the daughter graph instances, the bound value is used in creating new labels which either end in a 0 (`!c0`) or end in a 1 (`!c1`).

The first four applications of the transformation are shown in Figure 4.

For this cube grammar, local layouts did not perform well. Instead post-generation, user-assisted placement heuristics were employed. Currently our heuristics are modifications of an approach in which the graph is viewed as an assemblage of steel rings (the nodes) and springs (the edges) that is allowed to relax into a minimal energy configuration [10]. ParaGraph includes a standard interface for experimenting with placement algorithms and, eventually, we hope to provide a wide selection of heuristics.

The ParaGraph editor, thus provides usable interface to our graph grammar formalism. We have been careful to adhere to the formalism of aggregate rewriting in our construction of its graph grammar "engine". Furthermore, we have confined the implementation of graph rewriting to a module which could be easily replaced to support alternate characterizations of graph rewriting.

4. ParaGraph Extensions

ParaGraph, as described above, is often quite cumbersome as a parallel programming tool. In this section, we describe a number of extensions that have been included to facilitate programming.

Again, we proceed with a series of examples. For clarity and brevity, we omit scorecards and, instead, label graphs directly. We also omit step-by-step instructions on the use of ParaGraph and assume that the reader already has the flavor of its interface. Unless otherwise indicated, the layouts in this section were not generated automatically.

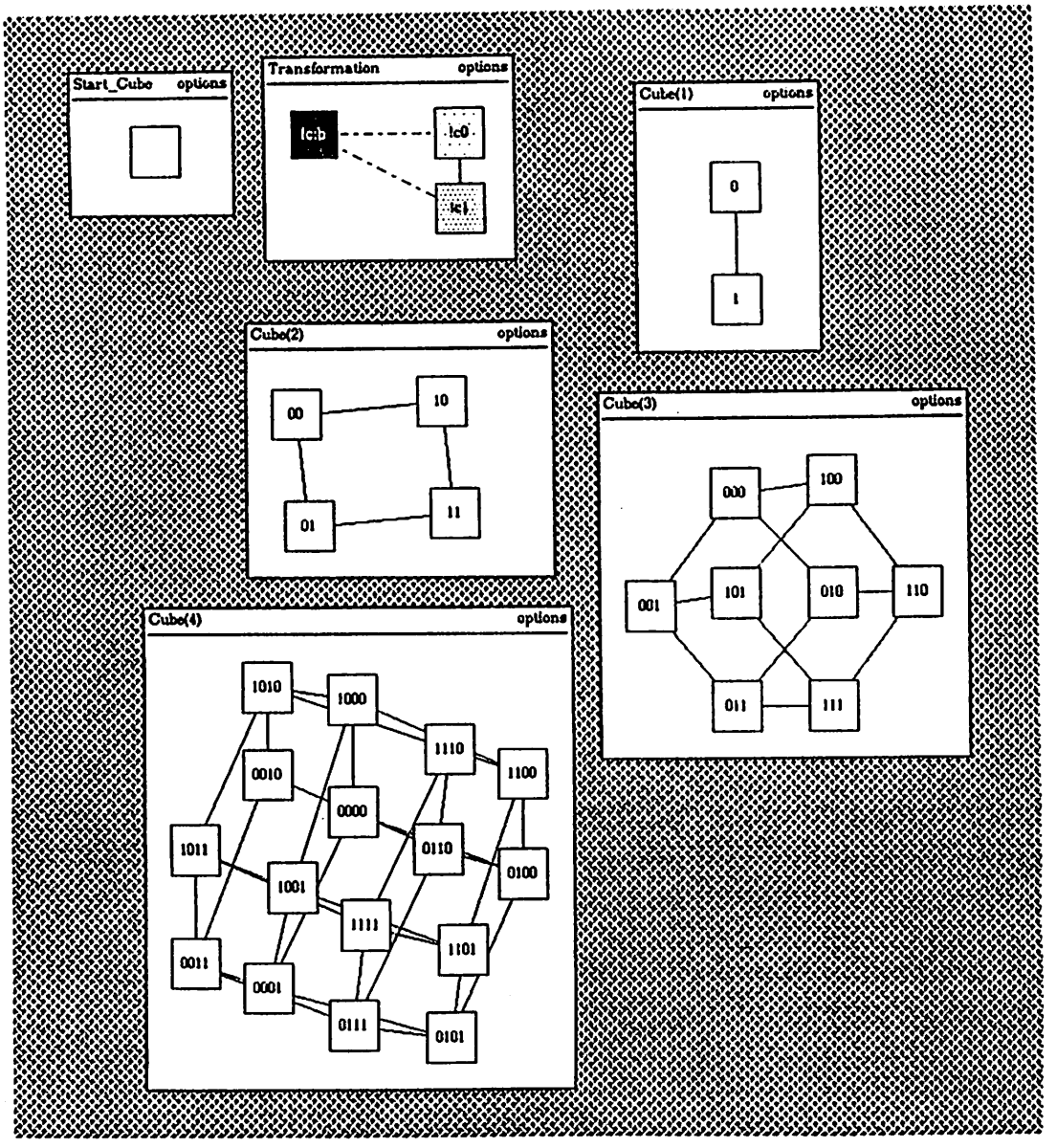
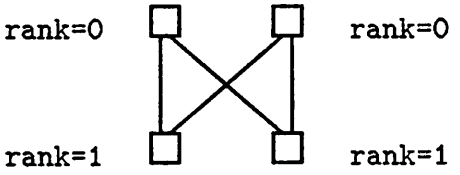


Figure 4: ParaGraph specification of the family of hypercubes together with the first four members of that family generated by the script `Start:(Transformation)n`.

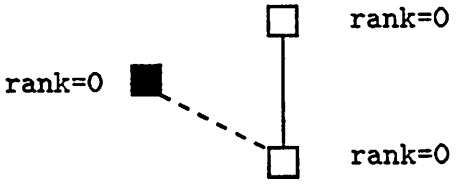
Example 3: Butterflies and Banyans. In this example, we introduce a mechanism for providing restrictions on the domain of a transformation.

To describe the family of butterfly graphs, we begin with a four node start graph annotated by a single, user-defined attribute giving its rank:

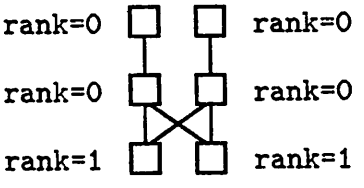


There are three transformations. The first begins a new rank of the butterfly by adding a row of nodes connected along the top level:

(B1)

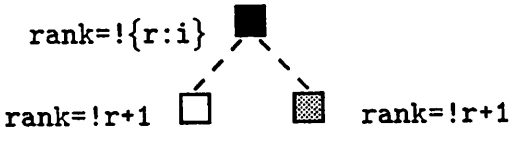


The transformation applies only to nodes in the top level because labels of matched mother graph instances must have rank=0. Its first application results in

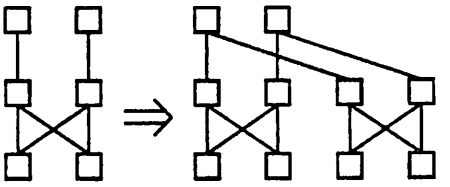


The second transformation:

(B2)



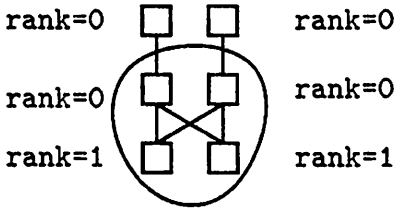
should make a connected copy of the original host graph by rewriting nodes on the bottom two ranks:



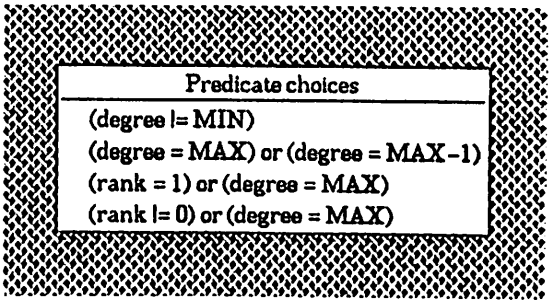
However, with the given labeling and ParaGraph's simple string matching facilities, it is not possible to restrict rewriting to just the lower ranks. Additional labels could be introduced but this is often cumbersome. Instead, ParaGraph allows the user to limit the domain of a transformation directly with an associated *restricting predicate*. The transformation is only applied to those nodes for which the associated predicate evaluates to TRUE.

Restricting predicates are most often specified by example: the user selects a subset of nodes from a sample graph and heuristics are used to convert his selection into a generalized, closed form expression [26,27]. Our heuristics generate predicates which are conjunctions and disjunctions of *base predicates* of the form *attribute* = *value* or *attribute* $\neq$ *value* (attributes can be limited to an "interesting" subset). Predicates are determined both for the selected subset and for its complement. They are generalized by replacing integer attributes with simple expressions involving MIN and MAX (the minimum and maximum values respectively over the current graph instance) where appropriate. A list of applicable predicates — ranked by generality and simplicity — are presented to the user who can either choose a selection from the list or enter his own alternative.

For this butterfly transformation, the user would select the two lower ranks of nodes



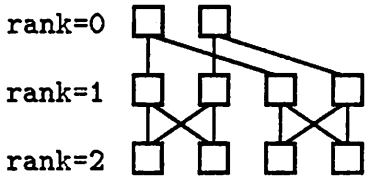
and the editor would generate the list of expressions:



We have found that these simple heuristics are quite adequate. In most cases, the highest ranked choice is an appropriate predicate for the transformation.

Once a restricting predicate is associated with a transformation, it is used whenever that transformation is applied, regardless of host graph characteristics.

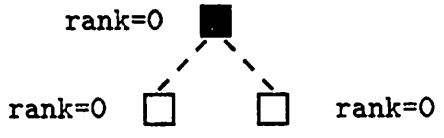
The first application of the restricted version of transformation B2 results in the graph



where all nodes at the same level have the rank attribute value shown to the left of the graph.

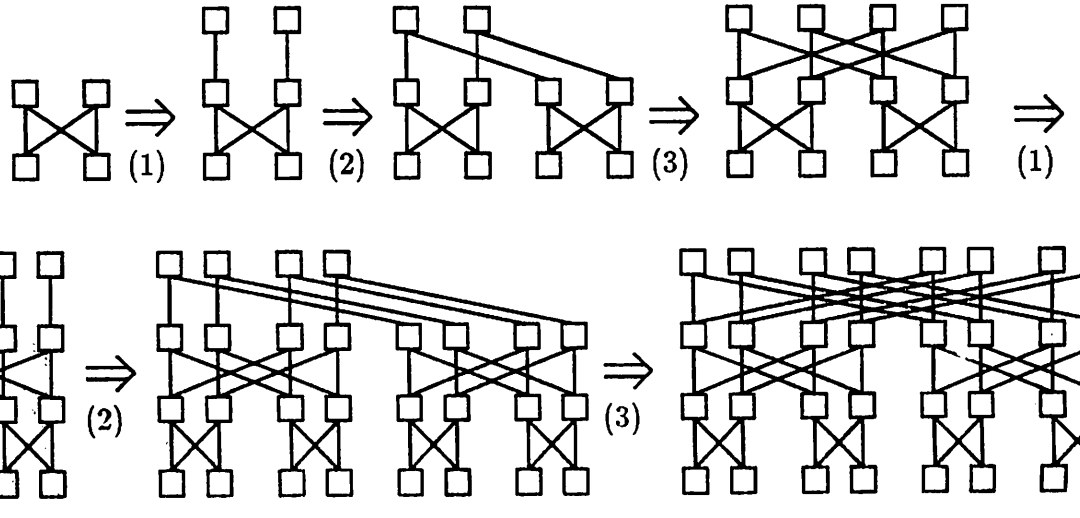
The third transformation:

(B3)



makes a connected copy of the nodes in the top rank to complete the butterfly. It does not use either partitioning or restricting predicates.

These three transformations can be iterated in the order specified to generate any butterfly:



If they are taken out of order, however, unwanted graphs occur. To control the derivation sequence, we define the BUTTERFLY script as

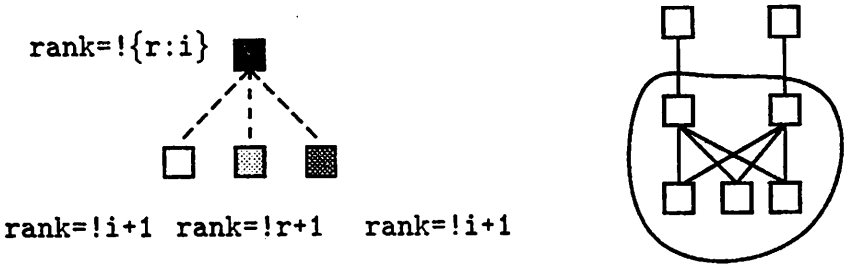
```
StartButterfly : (B1 B2 B3)n
```

where `StartButterfly` is the 4 node start graph from above. BUTTERFLY(2) and BUTTERFLY(4), then, are the 32 and 192 node butterflies respectively.

An alternate layout of the butterfly is shown in Figure 5. It was generated automatically and differs in orientation from the layout we have described above. Eventually, we expect to provide a menu of local layout options that will enable the user to have more control over placement.

Note that the transformations (B2) and (B3) of this grammar could be trivially modified to create any desired number of copies and thus they could be used in generating other SW-banyan networks [12]. In forming regular banyans, for example, the number of copies created by the second transformation controls the fan-in (f) and the number of copies created in the third transformation controls the spread (s). To specify the family of SW-banyans with $f = 3$ and $s = 2$, for example, we would have to modify the second transformation to

(B2')



where the selection needed to generate the appropriate restricting predicate is shown to the right. This new version of the transformation (together with an appropriately modified start graph) would result in the derivation

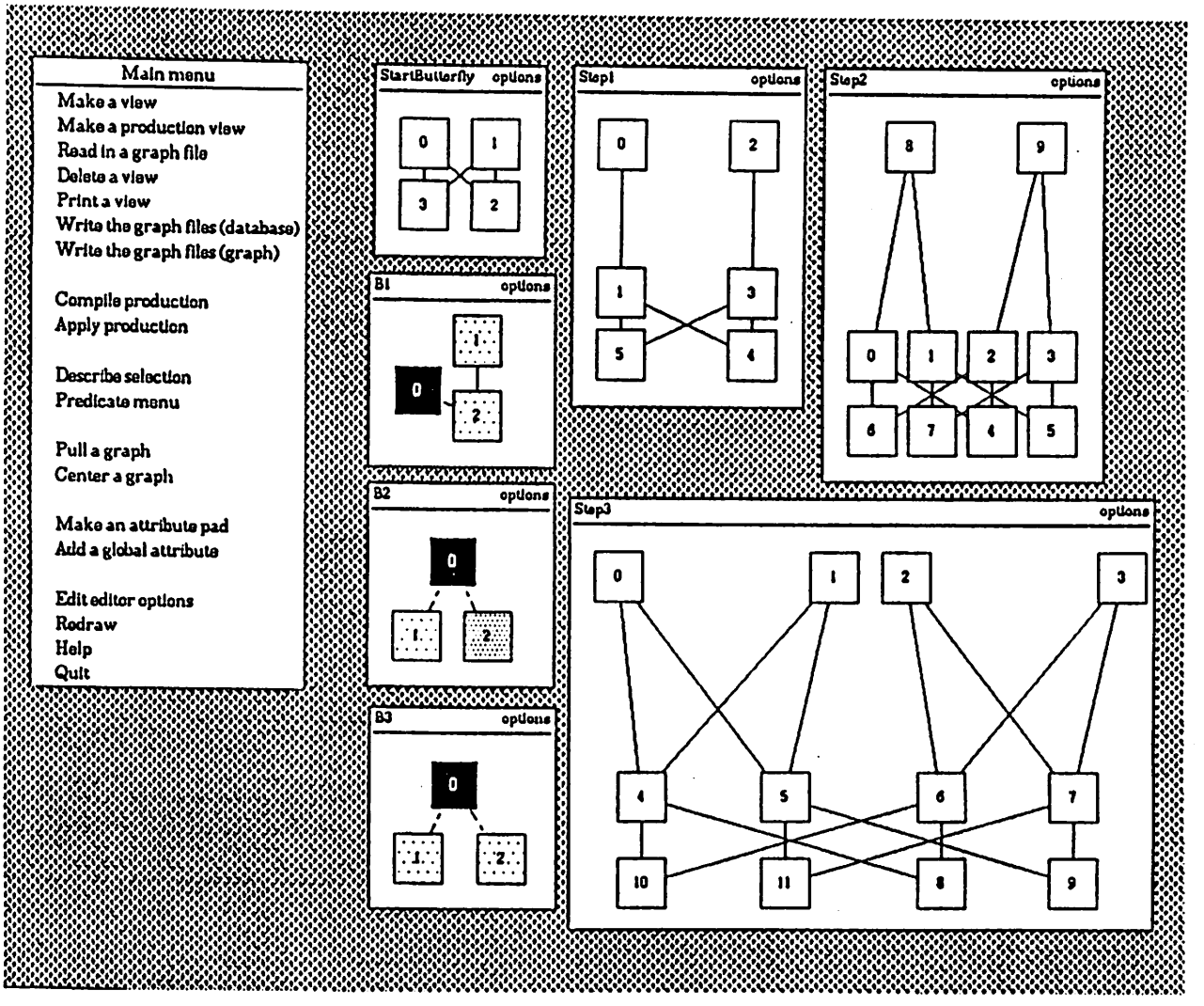
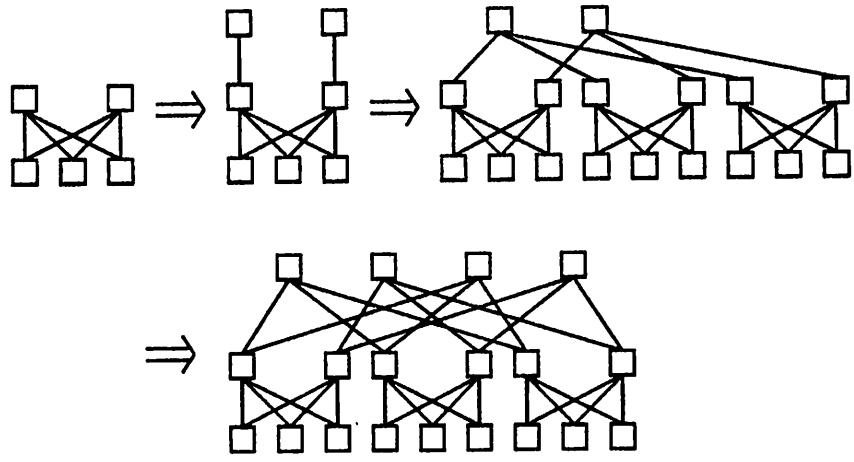


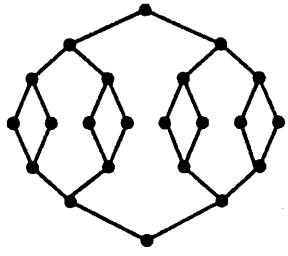
Figure 5: Butterfly grammar and the results of the first iteration of the script (Steps 1 through 3).



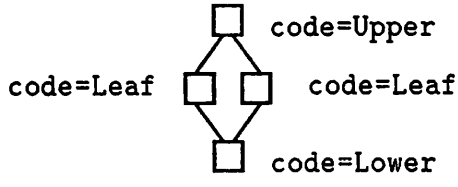
from the script `StartButterfly'` : (B1 B2′ B3)n.

Example 4: Database Trees. In this example, we introduce edge inheritance.

We specify a family of graphs, called database trees, that are formed by identifying the leaves of two complete binary trees:



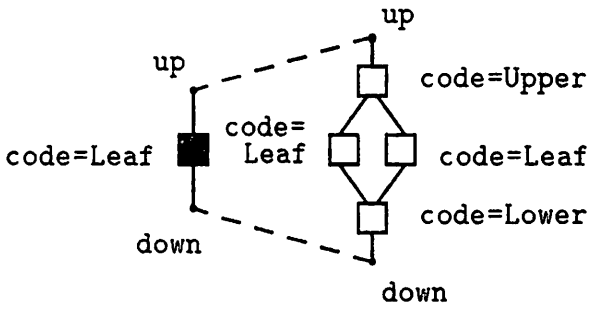
We begin with a four node start graph:



The obvious transformation — in which leaves are replaced with four node database trees — does not work, however, because the top root of the daughter graph needs to inherit just the upward connections of the leaf it replaces while the bottom root of the daughter graph needs to inherit just the downward connections of the leaf it replaces.

Using the basic AR rewriting mechanism, inheriting daughter graph nodes always inherit all of the edges of the associated mother graph node. ParaGraph refines this mechanism to allow partial inheritance: daughter graph nodes can inherit selected subsets of nodes. To distinguish these subsets with a minimum of context, we introduce the notion of a *junction* which is the incidence of an edge and a node. Each edge has two junctions. Junctions can be labeled with sets of attributes and their labels can be matched and rewritten in the same manner as node labels.

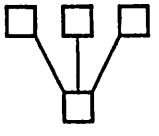
For the current example, the desired transformation with partial inheritance is



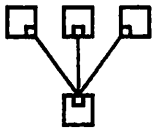
where the junctions are shown as partial edges labeled up and down.

This transformation indicates that the top node of the daughter graph inherits just those edges with junctions labeled up from the mother graph node and the bottom node of the daughter graph inherits just those edges with junctions labeled down. Junctions participate only partially in occurrence matching: where the corresponding edges are missing from the mother graph instances (as, for example, at the edges of a graph), they are not required for matching; where they exist, however, their labels are matched and possibly rewritten. This mechanism aids in uniform rewriting of graphs that are not entirely symmetric.

Note that the two junctions of an edge need not be labeled the same: junctions provide local views of edges. They are thus closely related to the concept of a *port* that has been used, for example, in Poker. In our view, a port is a set of similarly labeled junctions; it has no significance in the graph description but it may be relevant for particular applications. For these applications, "Show ports" has been included as a display option; if invoked, it would render the graph



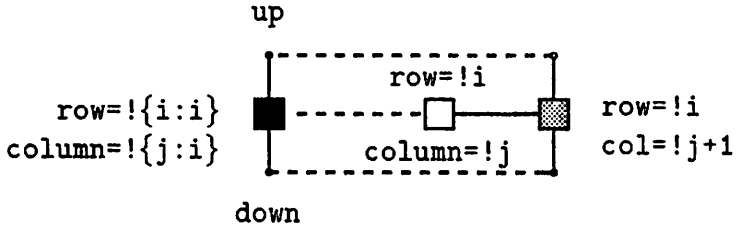
where all junctions on the lower node are similarly labeled, as

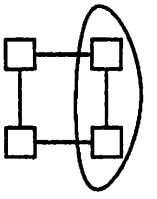


Example 5: Grids. In this example, we introduce edge partitioning.

There are a number of ways of describing the family of grids. Perhaps the most obvious is to define a four node instance and then repeatedly add columns and rows. Assuming that nodes are annotated by row and column attributes, the transformations to add a new column

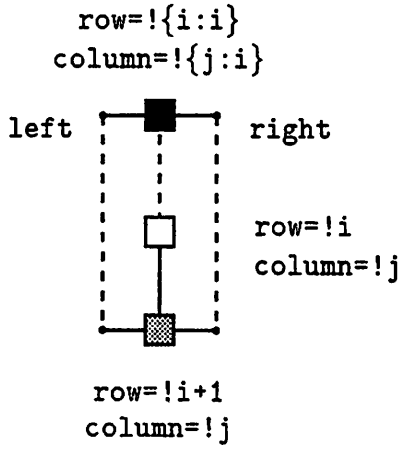
(G1)

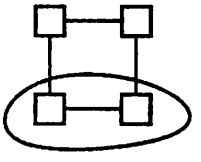




and to add a new row

(G2)





require restricting predicates (shown by the sample selections to the right of the transformations), as well as node inheritance (at the unshaded daughter graph nodes) and edge inheritance (at the shaded daughter graph nodes).

Since junctions missing from mother graph occurrences are ignored, all nodes — including boundary nodes which do not have the full complement of four neighbors — are rewritten with this same transformation. Thus, the script

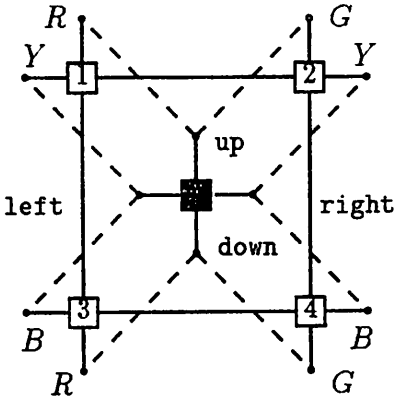
```
Start_Grid :(G1 G2)n
```

describes the set of square grids, while the script

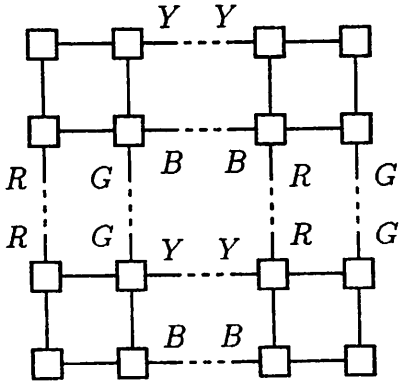
```
Start_Grid :(G1)n(G2)m
```

describes the set of $(n+2) \times (m+2)$ rectangular grids.

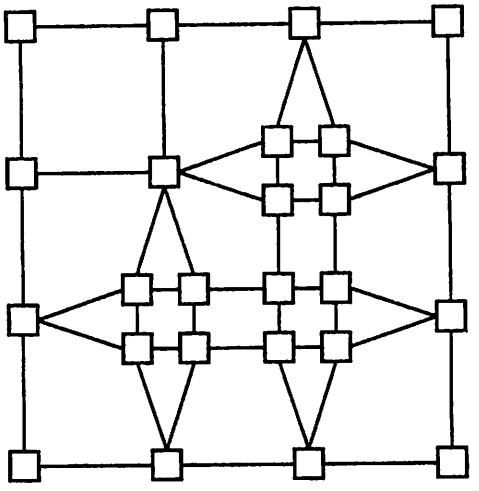
An alternate method of describing a square grid is to use recursion, expanding each node to a 2×2 subgrid. This transformation requires the introduction of edge partitioning. In a manner analogous to node partitioning, junctions are partitioned and inherited edges between instances of daughter graphs respect this partitioning. The required transformation is



where the central node is replaced ("expanded"). The junctions of the daughter graph are partitioned into four sets. As before, partitioning is indicated by color; for the purposes of this paper we use labels R, B, G and Y (for red, blue, green and yellow). Nodes corresponding to nodes 1 and 2 of the daughter graph, as an example, inherit upward edges that are connected to downward edges from nodes 3 and 4 of other daughter graph instances within the same partitions. Thus in transforming a 4 node grid into a 16 node grid, only the dashed connections are made:



The above transformation can be selectively applied to subsets of nodes to generate grids with non-uniform refinements:



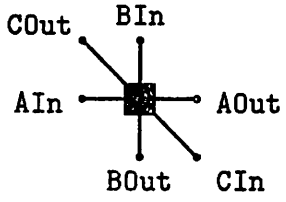
In a manner consistent with node inheritance, edges connecting refined grids to other refined grids respect partitioning while those connecting refined grids to unrefined nodes ignore partitioning.

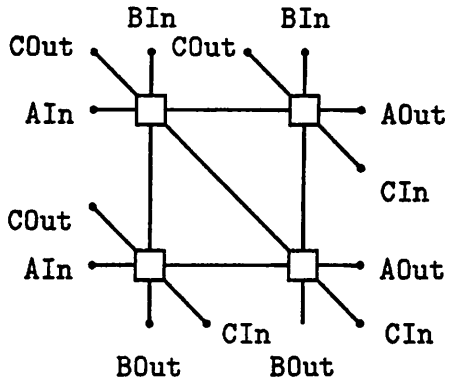
Refinements such as these are not easy to describe in conventional facilities for representing graphs. We expect that they will be important, especially in the future as we consider support for dynamic changes to graph structures.

Example 6: Hexagonal Mesh. In this example, we do not introduce any new features but instead demonstrate a more complicated instance of edge inheritance with partitioning.

Our specification of the family of hexagonal meshes [18] uses the same recursive technique used above — individual nodes are expanded to 2×2 submeshes. In this case, however, the required transformation is more complicated and, as a result, we show it in stages.

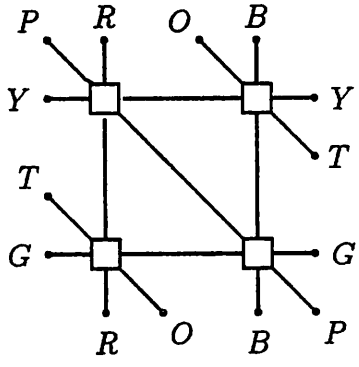
We begin by defining the mother and daughter graphs along with their junction labels:



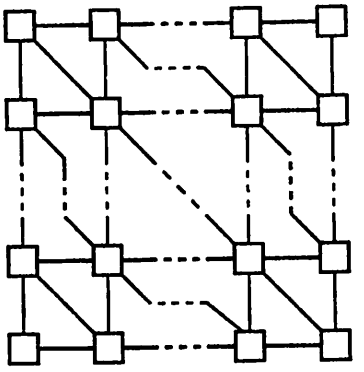


The definition of inheritance requires the introduction of 14 intergraph edges, connecting all pairs of similarly labeled junctions. Because it is difficult to display this many edges coherently, ParaGraph permits a hierarchical display: partial inheritance is indicated by dotted intergraph edges which can be individually expanded in order to specify their details with junction-to-junction edges. For the purposes of this paper, we assume that inheritance edges have been defined between all pairs of mother and daughter graph junctions having the same label.

To complete the transformation, we must partition the edges; using the letters R, Y, B, G, T, P, and O to denote colors, we have



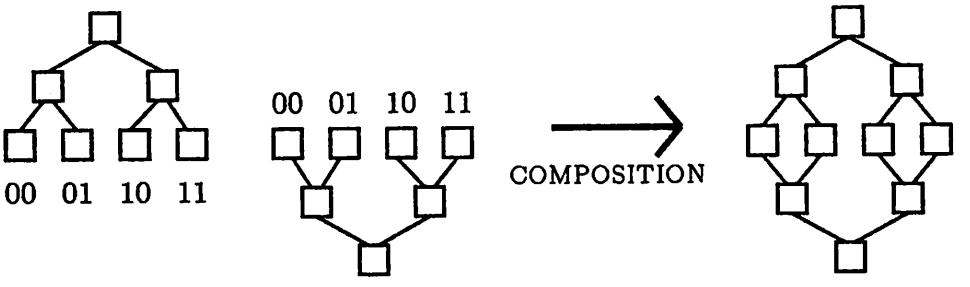
indicating that only the dashed connections are to be made:



(As this example illustrates, we have not resolved all of the issues inherent in the display of complex graph structures. We expect to be able to address these issues more satisfactorily as we accumulate experience with the editor.)

Example 7: Database Trees revisited. In this last example, we use an alternate specification of the database tree of Example 4 to introduce a mechanism for graph composition.

We construct the graph family tree as the composition of two families. We begin by specifying binary tree families for both the upper and lower trees (these specifications are obviously similar to that of Example 1 and we omit them). We next generate appropriately sized graph instances from each grammar separately and then "glue" them together at their leaves. Gluing identifies nodes with the same values for some set of attributes:



In identifying two nodes, we create a merged node having the union of their edge sets. The attribute values for the new node are determined by combining the values from the original nodes: where they agree, the common value is chosen (in most cases); where they disagree, a handler is invoked to resolve the discrepancy. Thus, for example, attributes representing parameters might be joined into a parameter list, attributes representing node degree might be recalculated, and attributes representing code segments might be concatenated into a program segment. A standard collection of handlers is available.

This same mechanism of node identification will enable users to form composite graphs for programs that include a variety of subcomputations.

ParaGraph thus extends the AR mechanism in a number of ways — adding restricting predicates, edge inheritance, and graph composition — to make it a more convenient tool for the programmer. In the next section, we briefly discuss ParaGraph's integration into a parallel programming environment.

5. Using ParaGraph in a Parallel Programming Environment

Our motivation for designing the graph editor was to provide support for the explicit representation of graphs for use within parallel programming environments. ParaGraph is central to the programming process in our environment [9]. Its output — in the form of both annotated graphs and underlying graph grammars — is accessible to tools supporting all phases of program development. We currently use it in three activities:

- *Specifying Parallel Programs.* Following the approach used in Poker [22], we view a parallel program as an annotated graph. Annotations might, for example, include code segments, run-time parameters, port associations, or compile-time constants. This form of programming reduces the disparity between a programmer's conceptualization of his algorithm and its implementation, increases the homogeneity of process code [2], and forms the basis for coherent graphical displays [15,23]. It is only feasible, however, if the program descriptions are scalable.

  ParaGraph permits the user to specify an annotated family of programs. Specific programs are generated as graph family instances and then preprocessed into an form appropriate for compilation. Currently, we produce a set of C programs and channel declarations suitable for execution by the Simon Multiprocessor Simulator [13,14] but extensions to other environments are straightforward.

- *Specifying Canister Communication.* For many massively parallel algorithms, interprocess communication is often quite structured: streams of related data are pipelined along sequences of channels, values are broadcast to (and aggregated from) sets of processes, or messages are routed through intermediate processes to their intended destinations. In each case, communications are best understood not as isolated point-to-point message transmissions, but as components of *global patterns of communication.* These patterns are fundamental to our understanding of parallelism but they are not supported by existing parallel programming environments.

Previously, we introduced a programming construct, called *canister communication*, that supports the explicit description of global communication patterns [3]. To use canister communication, the programmer must describe annotated paths through a process array, called *itineraries*. ParaGraph provides two mechanisms for these descriptions: the first defines a path through an existing process structure and the second defines an itinerary separately as a directed subgraph (implicitly defining the process structure as the composition of all such graphs) [26,27].

- *Parallel Debugging.* Parallel debugging for massively parallel, message-passing systems is extremely difficult: the programmer must contend with an overwhelming amount of potentially relevant information and he cannot rely on consistent global states or reproducible behavior. Often these systems are best approached by modeling their behavior: the programmer determines the extent to which models of the intended behavior correspond to models of the actual behavior [5].

  To facilitate this modeling for message-passing behavior, we have developed a pattern-oriented parallel debugger [15], called Belvedere.[7] Belvedere allows the user to define abstract communication events which are then automatically animated. Belvedere utilizes ParaGraph in two ways. Editor specifications of process interconnection structures are used in generating displays for animation and editor descriptions of itineraries are used in defining abstract communication events.

While we have just begun to integrate ParaGraph into our environment, it is clear that graphical specifications and representations are useful in a variety of support tools.

6. Status and Conclusions

ParaGraph is the first graph tool to provide the comprehensive range of support needed within a parallel programming environment. Most significantly, it provides scalability, allowing the user to concisely describe not just graph instances, but entire graph families. It also features flexible mechanisms for graph annotation and composition, and it supports graph visualization with a variety of layout heuristics. ParaGraph has been successfully integrated into a parallel programming environment where it is used by tools for program specification and debugging.

The editor, as described here, has been designed and its implementation is largely complete.[8] We continue, however, to investigate a number of issues, including the display

[7] *Belvedere* comes from the Latin *bellus* meaning "beautiful" and *vedere* meaning "view."

[8] At the time of this writing, the implementation of edge inheritance is the only significant section of code missing.

and storage of very large graphs, the identification of predefined graph transformations, the use of more powerful labeling mechanisms, and the introduction of additional facilities for graph manipulation. We are also investigating the generalization of our techniques to less regular graphs and to graphs that change dynamically.

Acknowledgements. We would like to thank a number of people for their contributions to ParaGraph. John Hagerman developed several early prototypes which influenced our design; Jim Ahrens, David Black, Nandakumar Varadaraju and Qing Yu contributed to the current design and implementation. Lawrence Snyder, a continuing source of ideas and suggestions, made extensive comments on earlier drafts of this paper.

References

1. Bailey, D. A. *Specifying Communication for Massively Parallel Ensemble Machines.* Ph.D. Thesis, University of Massachusetts, Amherst, September 1988.

2. Bailey, D.A. and Cuny, J.E. *Visual Extensions to Parallel Programming Languages.* Technical Report 89-69, COINS Department, University of Massachusetts, 1989.

3. Bailey, D. A. and Cuny, J. E. *Canister Communication in Parallel Programs.* Technical Report 88-42, COINS Department, University of Massachusetts, October 1988.

4. Bailey, D. A. and Cuny, J. E. Graph grammar based specification of interconnection structures for massively parallel computation. *Proceedings of the Third International Workshop on Graph Grammars*, Springer-Verlag, Berlin (1987), pp. 73–85.

5. Bates, P. C. and Wileden, J. C. High-level debugging of distributed systems: the behavioral abstraction approach. *Journal of System Software* 3 (1983), pp. 255–244.

6. Berman, F., Goodrich, M., Koelbel, C., Robison III, W. J., and Showell, K. Prep-P: a mapping preprocessor for CHiP architectures. *1985 International Conference on Parallel Processing* (August 1985), pp. 731–733.

7. Browne, J. C., Azam, M., and Sobek, S. CODE: a unified approach to parallel programming. *IEEE Software* (July 1989), 10–19.

8. Couch, A. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors.* Ph.D. Thesis, Tufts University, Medford, Massachusetts, 1988.

9. Cuny, J. E, Bailey, D. A., Hagerman, J. W., and Hough, A. A. The Simple Simon Programming Environment: A status report. *Proceedings of the Twenty-Fifth Annual Allerton Conference on Communication, Control and Computing* (September 1987), pp. 238-247.

10. Eades, P. A heuristic for graph drawing. *Congressus Numeratium* 42 (May 1984), pp.149–160.

11. Gisi, M., Cuny, J. E., and Bailey, D. A. Canister communication as a vehicle for parallel debugging. In *Proceedings of the First Annual IEEE Symposium on Distributed and Parallel Processing* (May 1989), pp. 198–199.

12. Goke, R.L. *Banyan Networks for Partitioning Multiprocessor Systems.* Ph.D. Thesis, University of Florida (1976).

13. Fujimoto, R. M. *SIMON: Simulator of Multicomputer Networks.* Technical Report UCB/CSD 83/140, University of California at Berkeley, August 1983.

14. Heller, D.E. *Multiprocessor Simulation Program SIMON.* Shell Development Corporation, 1985.

15. Hough, A. A. and Cuny, J. E. Initial experiences with a pattern-oriented parallel debugger. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988). Also appeared as *SIGPLAN Notices* 24(1) (1989), pp. 195–205.

16. Li, H., Wang, C. C., and Lavin, M. Structured Process: A new language attribute for better interaction of parallel algorithm and architecture. *Proceedings of the 1985 International Conference on Parallel Processing* (August 1985), pp. 247–254.

17. Jablonowski, D. and Guarna, V.A. GMB: A Tool for Manipulating and Animating Graph Data Structures. *Software - Practice and Experience* 19(3) (March 1989), pp. 283–301.

18. Kung, H. T. and Leiserson, C. Systolic arrays (for VLSI). In Mead, C. and Conway, L., editors, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

19. Newbery, F. J. An interface description language for graph editors. *Proceedings IEEE 1988 Workshop on Visual Languages*, 1988, pp. 10–12.

20. Nichols, K. M. and Edmark, J. T. Modeling multicomputer systems with PARET. *Computer* 21(5) (May 1988), 39–48.

21. Purtilo, J., Reed, D.A., and Grunwald, D.C. Environments for Prototyping Parallel Algorithms. *Proceedings of the 1987 International Conference on Parallel Processing* (August 1987), pp. 431–438.

22. Snyder, L. Parallel programming and the Poker programming environment. *Computer* 17(7) (July 1984), 27–37.

23. Socha, D., Bailey, M. and Notkin, D. Voyeur: Graphical views of parallel programs. *Proceedings of SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988), pp. 216-225.

24. Stallman, R. *GNU Emacs Manual* Emacs Version 18 for Unix Users. 1986.

25. Stiny, G. *Pictorial and Formal Aspects of Shape and Shape Grammars.* Birkhauser Verlag, Basel and Studgart, West Germany, 1975.

26. Yu, Q. and Cuny, J. E. Support for subgraph identification in a parallel programming environment. *Proceedings of the First IEEE Regional Symposium on Distributed and Parallel Processing* (May 1989), pp. 196–197.

27. Yu, Q. *Subgraph Identification in a Parallel Programming Environment.* Technical Report 89-30, COINS Department, University of Massachusetts, 1989.